Run-time Environment for the SARL Agent-Programming Language: the Example of the Janus platform

Stéphane GALLAND[a], Sebastian RODRIGUEZ[b], Nicolas GAUD[a]

[a]*LE2I, Univ. Bourgogne Franche-Comté, UTBM, F-90010 Belfort, France*
[b]*GITIA, Universidad Tecnológica Nacional, San Miguel de Tucumán, CPA T4001JJD, Argentina*

Abstract

SARL is a general-purpose agent-oriented programming language. This language aims at providing the fundamental abstractions for dealing with concurrency, distribution, interaction, decentralization, reactivity, autonomy and dynamic reconfiguration that are usually considered as essential for implementing agent-based applications. Every programming language specifies an execution model. For SARL, this run-time model is supported by a SARL run-time environment. The goals of this paper are to highlight the key principles for creating a SARL run-time environment, and its concrete implementation into the Janus agent platform.

Keywords: SARL agent-programming language, Run-time Environment, Janus platform

---

1. Introduction

In past years, multi-agent systems (MAS) have taken their place in our society. Application fields include robotics, artificial intelligence, cinema, video games. This evolution is the answer to increasingly complex projects, which require “intelligent” systems. Multi-agent systems allow to implement solutions with intelligence, capable of reasoning, learning and interacting between different agents. These systems represent a totally different way of looking at things. This way of designing systems resulted in new tools, methodologies and architectures, better suited to MAS modeling, e.g. ASPECS [1], MaSE [2] or even Gaia [3]. These methodologies are complemented by agent platforms for supporting the run-time execution of the designed models. There are several dozens, e.g. Jade [4], NetLogo [5], GAMA [6], or Janus [7].

*Corresponding author
Email address: `stephane.galland@utbm.fr` (Stéphane GALLAND)

On one hand, these solutions are highly interesting because they frame and provide tools for the development of agent-based systems. On the other hand, these systems are very complex to implement, and the conventional programming languages (Java, etc.) are not suited. There is therefore a real need for programming languages dedicated to MAS, which would offer more clarity to developers, and simplify developments. Several agent-programming languages have been proposed. Most of them are domain-specific languages, e.g. GAML [6], MARS [8], or Jason [9]. Several of these languages are general-purpose, e.g. SARL [10]. SARL [10] is a new general-purpose agent-oriented programming language (APL). SARL tries to set up adaptive and modular principles for developing multi-agent systems.

Every programming language specifies an execution model, and many implement at least part of that model in a runtime system [11]. For a SARL program being executed, a specific run-time environment must be defined. *The goals of this paper are to highlight: (i) the key principles for creating a SARL run-time environment, and (ii) its concrete implementation within the Janus agent platform [7].* Neither the relationship between the above methodologies and SARL, nor the one between these methodologies and Janus are the purpose of this paper. Our motivations are:

i) to enable Researchers, Engineers and Students to create a run-time environment for SARL. The key principles and features to be considered for designing this run-time environment are provided to them; and

ii) to give a proof of concept based on the Janus platform. This platform already provides an implementation for holonic multi-agent systems, which are also a core concept within SARL. In this paper, the Janus meta-model and software architecture are updated in order to fit the SARL requirements.

This paper is structured as follow. Section 2 explains the fundamentals of the SARL language. Section 3 presents the compilation and execution tool-chain for SARL. Section 4 details the key principles for creating a SARL run-time environment that handles any SARL program. Several existing agent platforms are presented in Section 5. Section 6 gives details on the re-implementation of the Janus platform in order to create a SARL run-time environment. Section 7 describes the performance evaluation of the Janus implementation. Finally, Section 8 concludes this paper and provides perspectives to this work.

2. SARL Agent-Programming Language

SARL[1] is a general-purpose agent-oriented programming language [10]. It aims at providing the fundamental abstractions for dealing with standard agent

[1] Official website: `http://www.sarl.io`

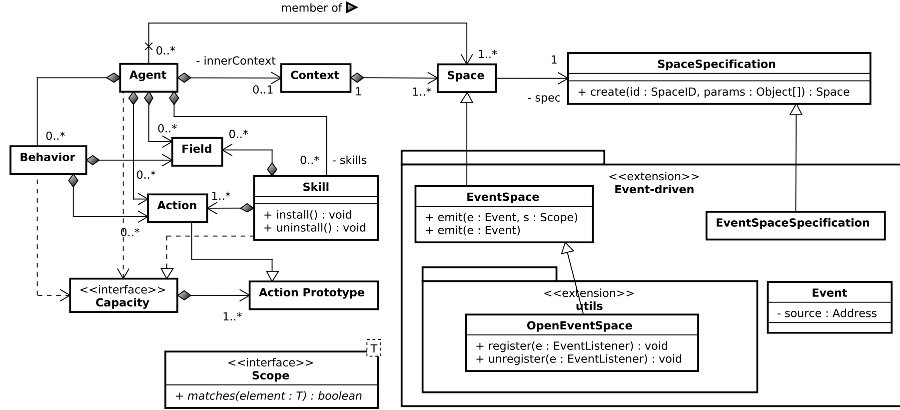

Figure 1: UML Class Diagram defining the major concepts in the SARL meta-model.

features: concurrency, distribution, interaction, decentralization, reactivity, autonomy and dynamic reconfiguration. The main perspective that guided the creation of SARL is the establishment of an open and easily extensible language. Such language should thus provide a reduced set of key concepts that focuses solely on the principles considered as essential to implement a multi-agent system. The major concepts of SARL are explained below, and illustrated in Figure 1.

2.1. Action

An action is a specification of a transformation of a part of the designed system or its environment. This transformation guarantees resulting properties if the system before the transformation satisfies a set of constraints. An action $a \in \mathbb{A}$ is defined by its prototype and its body, as illustrated by Equation 1. The prototype is composed by the name n_a of the action, a sequence P_a of formal parameters, and the type r_a of the returned values. The body B_a is a sequence of expressions — a subset of the language constructs for describing evaluable expressions — that represents transformations. These two parts of the action's definition are represented respectively by the `Action Prototype` and `Action` in Figure 1.

$$a = \langle n_a, P_a, r_a, B_a \rangle \tag{1}$$

For pedagogical reasons, the SARL action concept could be linked to method concept into the object-oriented paradigm: both concepts represent the same language construct.

2.2. Capacity and Skill

A capacity $c \in \mathbb{C}$ is the specification of a collection P_c of actions' prototypes, as defined in Equation 2. A capacity could be used to specify what an agent

can do, and what a behavior requires for its execution.

$$c = \langle S_c, P_c, F_c \rangle \tag{2}$$

This specification makes no assumptions about its implementation. So that, the bodies of the actions are not defined into the capacity: $\forall p \in P_c \subseteq \mathbb{A}, B_p = \emptyset$. The capacity c could inherit a part of its definition from another capacities S_c. The fully expanded set F_c of actions that are defined into the capacity c is defined by: $F_c = P_c \cup \{F_s / s \in S_c\}$.

A skill $s \in \mathbb{S}$ is a possible implementation of capacities C_s fulfilling all the constraints of these specifications, as defined by Equation 3.

$$s = \langle s_s, C_s, A_s, F_s \rangle \tag{3}$$

A_s is the set implementations of the actions from the capacities C_s, such that $\forall a \in A_s, B_a \neq \emptyset$ and $A_s \subseteq \bigcup_{c \in C_s} F_c$. Similarly to a capacity, a skill s could inherit a part of its definition from another skill s_s. The fully expanded sequence F_s of actions that are defined into the skill s is defined by: $F_s = A_s \cup F_{s_s}$. Skill s provides an implementation for each capacity action: $a \in F_s / \forall a \in A_c, \forall c \in C_s$.

An agent can dynamically evolve by learning or acquiring new capacities. It can also dynamically change the skill associated to a given capacity [12, 1]. Acquiring new capacities also enables an agent to get access to new behaviors requiring these capacities. This provides agents with a self-adaptation mechanism that allow them to dynamically change their architecture according to their current needs and goals.

2.3. Context and Spaces

A space $p \in \mathbb{P}$ is the support of the interaction between agents, respecting the rules defined in an associated space specification. SARL natively defines a particular type of space, namely the *event space* to provide a support to event-driven interactions. Within an event space, agents communicate using events. Nevertheless, it is possible to define programmatically new types of spaces that are not event-based. Equation 4 provides the definition of a space p. M_p is the set of agents, which are participating to the interaction within the space, i.o.w. the members of this space. R_p is the functional definition of the interaction mechanism that is supported by the space. It may be a routing algorithm of messages between the space's participants, or the interaction specification among agents and artifacts [13].

$$p = \langle M_p, R_p \rangle \tag{4}$$

A context $c \in \mathbb{O}$ defines the perimeter/boundary of a sub-system, and gathers a collection S_c of spaces, as defined in Equation 5.

$$c = \langle S_c, d_c \rangle \tag{5}$$

Since their creation, agents are incorporated into a context called the *default context* (see the upper part of Figure 2, level n). The notion of context makes

complete sense when agents are considered composed or holonic (see Section 2.6 for details).

In each context, there is at least one particular space $d_c \in S_c$, called *default space*, to which all agents in context c belong. This ensures the existence of a common shared space to all agents in the same context. Each agent can then create specific public or private spaces to achieve its personal goals.

The concept of environment, as defined by Weyns et al. [14] could be linked to the concept of context in SARL. The spaces into a context become views to the environment [15]. Consequently, spaces restrict how agents perceive and act in the environment. Rodriguez et al. [16] have proposed to model the environment by defining a context, a space, and specific agents that are supporting the endogenous dynamics of the environment. This last type of agent is mandatory because there is no syntactic way within SARL for defining a context: only spaces and agents could be programmed.

When an agent is created, it belongs to a context, named its *default context*. During its life, an agent may join or leave other *external contexts*, as illustrated by Figure 2 with agent A. The invariant condition is each agent belongs to a default context, whenever this default context is not the one in which the agent was created. Figure 1 illustrates this membership relation by the "member of" association between Agent and Space. In other words, an agent belongs to a context if, and only if, it is member of the default space of this context. The direct relation between the agent and context concepts is related to the inner context, which is detailed in Section 2.6.

2.4. Agent and Behavior

An agent $a \in \mathbb{T}$ is an autonomous entity having a set B_a of behaviors, and a set S_a of skills to realize the capacities it exhibits. It is defined in Equation 6, and represented by the `Agent` type in Figure 1.

$$a = \langle B_a, S_a, M_a, d_a, C_a, i_a \rangle \tag{6}$$

Agent a has a set S_a of individual skills that may be used for building the agent's behaviors. Agent a defines the mapping $M_a : \mathbb{C} \to \mathbb{S}$ from one capacity to a single skill implementation. From this definition, the agent a is able to determine which skill should be used when a capacity's action is invoked. A set of capacity–skill pairs, named the built-in capacities (BIC) is defined into the SARL specifications [10]. They are considered as essential to respect the commonly accepted competences of agents, such autonomy, reactivity, proactivity and social capacities. The full set of BICs are presented in Section 4 because they must be implemented into, and provided by the SARL run-time environment.

Among these BICs, the `DefaultContextInteractions` and `ExternalContextAccess` capacities are defined. They give respectively the access to the agent's default context d_a, and the set C_a of contexts in which the agents belong to, such that $d_a \in C_a$. i_a represents the internal context of the agent, which is detailed in Section 2.6.

Another BIC is the `Behaviors` capacity. It enables an agent to incorporate a collection $B_a \subset \mathbb{B}$ of behaviors that will determine its global conduct. *A behavior $b \in \mathbb{B}$ maps a collection of perceptions represented by events to a sequence of actions.* $O_b : \mathbb{E} \rightarrow \mathcal{P}(\mathbb{A})$ is the mapping function in Equation 7.

$$b = \langle O_b \rangle \tag{7}$$

An agent has also a default behavior directly described within its definition. It is illustrated by the relationship between `Agent` and `Action` types in Figure 1.

By default, the various behaviors of an agent communicate using an event-driven approach. *An event $e \in \mathbb{E}$ is the specification of anything that happens in a space s, and may potentially trigger effects by a listener*, e.g. agent, behavior.

2.5. Example of SARL Program

For clarity reasons, let the definition of an agent, named `FactorialAgent` that is able to compute a factorial. This example is simple enough for illustrated the basics properties and features of the SARL language. Nevertheless, more complex implementation of multi-agent systems with SARL could be found in [17, 18, 19, 20, 21, 22, 23, 24].

`FactorialAgent` agent waits for other agent's request to calculate (`Calculate` event) a factorial. Once computed, it is notifying the result using the `ComputationDone` event.

```
event Factorial {
2   var upto : int
    var number : int
4   var value : int
}
6 event Calculate {
    var number : int
8 }
event ComputationDone {
10  var result : int
}
12 agent FactorialAgent {
    uses Lifecycle, Behaviors, DefaultContextInteractions
14  on Factorial [occurrence.number < occurrence.upto ] {
      wake(new Factorial => [
16          upto = occurrence.upto
            number = occurrence.number + 1
18          value = occurrence.value * (occurrence.number + 1)
        ])
20  }
    on Factorial [ occurrence.number == occurrence.upto ] {
22    emit(new ComputationDone => [ result = occurrence.value ])
      killMe
24  }
    on Calculate {
26    wake(new Factorial => [
            upto = occurrence.number
28          number = 0
            value = 1
30      ])
    }
32 }
```

Listing 1: Computation of Factorial with a SARL Agent

An agent is declared with the **agent** keyword (line 12). In the agent's body block, we can declare mental states (in the form of attributes), actions (or functions and event handlers). Actions that an agent can perform could be specified by capacities or natively inside the agent definition. Keyword **uses** imports the actions defined in capacities, so that, they can be accessed directly as an agent native function.

Agent perceptions and the sequence of actions the agent wants to perform for each perception are defined. This is achieved using the clause **on** <`perception`> [<`guard`>] {<`body`>}. `FactorialAgent` declares three behavioral event handlers (lines 14, 21, and 25). Perceptions for SARL agents take the form of events, and they can be declared using the **event** keyword. For instance, the `Calculate` event is defined at line 6. An event can carry information, in our case the number we want the factorial for.

When the `Calculate` event is perceived (line 25), the agent can access the event's instance using the **occurrence** keyword. At line 27, it sets the `upto` attribute using the information for the `Calculate` event occurrence.

From this point, the agent starts computing the factorial. The `Behaviors` built-in capacity provides the agent with mechanisms to (un)register new behaviors, and fire new internal events (`wake` action). For calculating the factorial, the agent fires an internal event of type `Factorial` using the `wake` action.

Two behaviors are declared for `Factorial` event (lines 14 and 21). When an event is perceived, SARL agents execute all their behaviors for that event type concurrently. Behaviors can declare guards to prevent their execution if required. So, the behavior at line 14 is only executed if **occurrence**.`number` < **occurrence**.`upto` evaluates to **true**. This behavior simply calculates the factorial for the next integer, and fires a `Factorial` event again. As illustrated, the guard's expression may reference the received event or any of its attributes. Nevertheless, any variable that is declared into the enclosing type (agent, behavior, etc.) of the guard, or any statically accessible variable or function may be referenced too.

Likewise, when the factorial for the requested number (stored in `upto` attribute) is found, the behavior at line 21 is executed. The `emit` action fires an event in the *default space* of the *default context* for notifying the computation is finished. After that, the agent stops its execution using the `killMe` action from the `Lifecycle` capacity.

It is necessary to clearly understand the difference between `wake` and `emit` actions. `Wake` fires an internal event within the agent that may be perceived by its own behaviors, and its members when it is composed by other agents. `Emit` action enables to fire an event in a given space that is outside the agent itself.

2.6. Recursive Agent and Hierarchical Multiagent System

In 1967, Koestler coined the term *holon* as an attempt to conciliate holistic and reductionist visions of the world. A holon represents a part–whole construct that can be seen as a component of a higher level system or as whole composed of other holons as substructures [25]. Holonic systems grew from the need to find

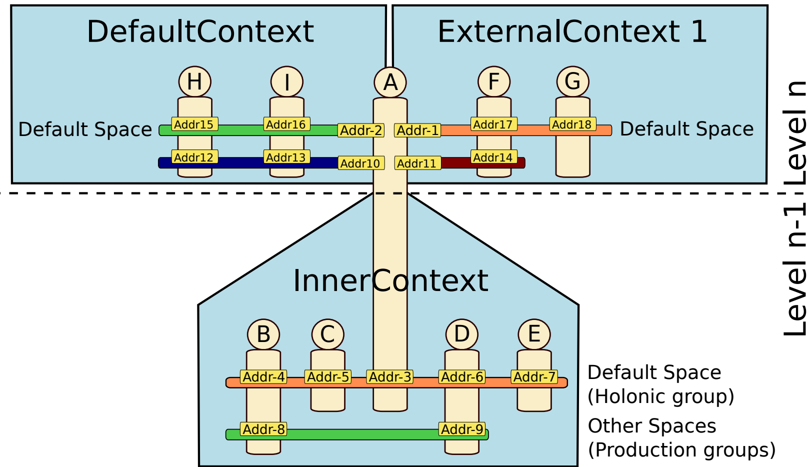

Figure 2: A Holon or a recursive agent in SARL

comprehensive construct that could help to explain social phenomena. Since then, it came to be used in a wide range of domains, including philosophy [26], manufacturing systems [27], and multi-agent systems [28].

Several works have studied this question, and they have proposed a number of models inspired from their experience in different domains. In many cases, the idea of *agents composed of other agents* could be found. Each researcher gives a specific name to this type of agent. Ferber [29] discusses *individual* and *collective* agents. *Meta-agents* are proposed by Holland [30]. *Agentified Groups* are taken into account in the works of Odell et al. [31]. All of these are examples of how researchers have called these "aggregated" entities that are composed of lower level agents. More recently, the importance of holonic MAS has been recognize by different methodologies such as ASPECS [32] and O-MASE [33].

In SARL, we recognize that agents can be composed of other agents. Therefore, SARL agents are in fact holons that can compose each other to define hierarchical or recursive MAS, called holarchies. In order to achieve this, SARL agents are structures that compose each other via their contexts. Each agent defines its own internal context, called *inner context* and it is part of one or more *external contexts*. For instance, in Figure 2, agent A is taking part of two *external contexts*, i.e. `Default Context` and `External Context 1`. The same agent has its own *inner context* where agents B, C, D and E evolve. Because, an agent may belong to a default and an external context at the same time, the resulting structure in not a simple hierarchy of agents (agent composed by agents). It is a directed graph of agents, with membership relations among them. In the SARL meta-model (Figure 1), these relations are formally supported by the "Member of" relation between an agent and a space.

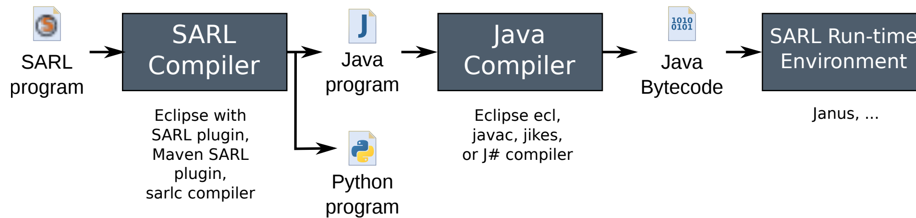

Figure 3: Compilation and Run-time Toolchain for SARL

3. SARL Tool-chain

The SARL tool-chain is the set of programming tools that are used to perform a multi-agent system with SARL. As illustrated by Figure 3, three types of tools are used in sequence in order to create and run an agent-based system:

- **SARL Compiler:** The SARL compiler transforms the source SARL language to the target language. Several target languages may be considered by this compiler. Because most of the agent frameworks are written with the Java language, the SARL compiler targets this object-oriented programming language by default, but not restricted to (a Python generator is also provided as proof-of-concept). The SARL compiler translates SARL statements into their object-oriented equivalent statements. Three different implementations of the SARL compiler are provided: a specific command-line tool (`sarlc`), the Eclipse development environment plugin, and a Maven plugin.

- **Java Compiler:** The files generated by the SARL compiler are standard Java files. They must be compiled with one of the standard tools that are available: eclipsec[2], javac[3], gcj[4], or jikes[5]. The result of the Java compilation is a collection of binary files (a.k.a. byte-code files) that may be run by a virtual machine.

- **SARL Run-time Environment:** The SARL Run-time Environment (SRE) is a collection of tools that enables running of an agent-based application written with SARL. Such an SRE must provide the implementation for each service and feature that are assumed to be provided by the run-time environment. Usually, a Java-based SRE is composed by the Java Run-time Environment (JRE) and a Java framework that supports the execution of the agents upon the JRE.

[2] Eclipse eclipsec: `https://www.eclipse.org`
[3] Oracle JDK: `http://www.oracle.com/technetwork/java/javase/overview/index.html`
[4] GNU Java Compiler: `http://gcc.gnu.org`
[5] IBM jikes: `http://jikes.sourceforge.net`

In order to create a valid SRE, it is mandatory to define the core features that are expected into a SRE implementation. Section 4 describes the key elements that should be considered for solving this specific point.

In order to give a Proof-of-Concept, a specific SRE implementation based on the Janus platform [7] is provided in Section 6. Janus is a Java application, i.e. a Java virtual machine is used for running the program upon a specific Java library that provides the features dedicated to the SARL agents. Janus was selected because, since its creation in 2008, it provides the key features for implementing holonic multi-agent systems. Therefore, it is the best candidate for becoming a SRE.

4. Key Points for SRE Creation

In this section, the major key points that should be considered for creating a SARL Run-time Environment are explained: support to the agent's life-cycle, and implementation of the built-in capacities.

4.1. Agent's Lifecycle

SARL does not imposes a specific agent's control loop. Indeed, when agents are spawned, the SRE is in charge of creating the agent instance and installing the skills associated to the built-in capacities into the agents. Then, when an agent is ready to begin its execution, SRE fires an `Initialize` event occurrence. This occurrence contains the initialization parameters for the agent's instance. Likewise, when the agent has decided to stop its own execution (using the `killMe` action from the `Lifecycle` capacity), SRE fires an `Destroy` event occurrence. It enables the agent to release any resource it may still hold. It is important to notice that agents cannot kill other agents, not even those that they have spawned. One of the key characteristics of an agent is its autonomy. From its definition, no other agent should be able to stop its execution without its consent. The MAS designer is free to implement any control or authority protocol for their own application scenarios.

4.2. Built-in Capacities

Every agent in SARL has a set of built-in capacities (BIC) considered essential to respect the commonly accepted competences of agents. These capacities are considered the main building blocks on top of which other higher level capacities and skills can be constructed. They are defined in the SARL language specifications, but the skills implementing them are provided by the SRE. This latest is in charge for creating the BICs' skills, and injecting them into an agent, before its execution begins. Therefore, when the agent receives the `Initialize` event, they are already available. The current eight defined BICs, and the actions they provide along their action signatures are:

- `ExternalContextAccess` provides access to the contexts that the agent is a part of, and the actions required to join and leave new contexts. The external context of the agent `A` is shown at the top right part of Figure 2.

- `InnerContextAccess` provides access to the *inner context* of the agent. This is keystone for holonic agent implementation. The inner context of the agent `A` is shown at the bottom part of Figure 2.
- `Behaviors` As previously described, agent can dynamically (un)register behaviors and trigger them with the `wake` action. This capacity is closely related to the `InnerContextAccess` to enable a high-level abstraction on holonic MAS development.
- `Lifecycle` provides actions to spawn new agents on different external contexts (peers), and the *inner context* (as holonic members). It also provides the `killMe` action to stop the agent execution.
- `Schedules` enables the agent to schedule tasks for future or periodic execution.
- `DefaultContextInteractions` is actually provided for convenience. It assumes that the action is performed on the agent's *default context* (upper left part of Figure 2) and its *default space*. For instance, the `emit` action is a shortcut for `defaultContext.defaultSpace.emit`(...). Therefore, it is actually created on top of the other BICs.
- `Logging` provides the agents for writing messages on the agent's log. The messages may be shown according to a severity level, e.g. information, warning, error.
- `Time` provides the agents for accessing to the current time. The time may be the operating system time, or a simulation time, depending on the implementation of the BIC within the SRE.

4.3. Parallel Execution

In most of the agent frameworks, e.g. Jade [4] and Janus [7] (before its adaptation to SARL), each agent is run on a separate execution resource, i.e. a thread. This design choice enables each agent managing its own execution resource, and participates to its autonomy. On several other platforms, e.g. TinyMAS[6], the agents are executed in turn in a loop. The parallel execution of the agents is therefore simulated.

SARL encourages a massively parallel execution of agents and behaviors. An agent entry point is a part of the agent behavior that is invoked from the outside of the agent. In SARL, the entry points are the behavior event handlers, specified by the **on** keyword. Each of these entry points is associated to a separate thread. Parallel execution of the pro-active behaviors of an agent is supported by the tasks that are launched with the `Schedules` built-in capacity.

Whatever the agent execution mechanism used by a SRE (thread-based or loop-based), a SARL developer always assumes that the agent's entry points are executed in parallel when he is writing the event handlers.

[6] TinyMAS Platform: `http://www.arakhne.org/tinymas`

5. Selection of an Agent Framework as SRE Candidate

SARL language specifies a set of concepts and their relations. It defines on top of them a collection of Built-In Capacities for agents. However, the SARL project does not impose a particular execution infrastructure. We consider that many different implementations of these concepts can be provided, and it can help SARL be developed faster. In this section, four Java-based agent platforms are considered as candidates for implementing a SRE: Jade [4], NetLogo [5], GAMA [6], and Janus [7].

According to the key points that are described in Section 4, several criteria are used to compare these agent frameworks:

- **Application type:** Multi-agent systems have a very wide scope of application, including 3D simulation, geography simulation, social simulation, video game.
- **Agent Type Definition:** Definition of a new agent may need to extend an existing feature or use specific language statements.
- **Agent Behavior Definition:** Definition of a new agent behavior may need to extend an existing feature or use specific language statements.
- **Interaction Mechanism:** This criterion indicates the type of interaction mechanism that is used by the agent framework: message exchanges, event-based interaction, stigmergy...
- **Agent Initialization:** This criterion indicates how an agent may be initialized.
- **Agent Destruction:** The destruction of an agent causes the release of resources. This criterion specifies the means for implementing a release function.
- **Agent Execution:** Agent execution mechanism is a key module of the agent platform. The type of mechanism may be synchronous; i.e. agents are run in a loop, or multi-threaded.
- **Hierarchical Multiagent System:** Hierarchical systems, and specifically holonic systems are a key principle behind the SARL agent programming language. This criterion indicates if, and how, a hierarchy of agents is supported by the agent framework.

Table 1 provides an overview of our agent framework comparison. Jade and Janus are both general purpose platforms: they could be used for building any agent-based application, including embedded system and simulation applications. NetLogo and GAMA are simulation platforms because they enables to build simulation software. We advocate that a general purpose platform fits better the needs for the SARL agents. Indeed, SARL does not target a specific application domain. It is defined for enabling the building of embedded and simulation-based applications, for instances.

SARL Key Concepts	Jade [4]	NetLogo [5]	GAMA [6]	Janus [7]
Application Type	general purpose	simulation	simulation	general purpose
Agent Type Definition	extension of `Agent` type	definition of species, no public Java API	definition of species	extension of `Agent` type
Agent Behavior Definition	extension of `Behavior` type	-	-	organizational model [32]
Interaction Mechanism	mailbox with messages	stigmergy	mailbox with messages	mailbox with messages
Agent Initialization	overriding of the `setup` function	specific statement `to setup`	overriding of the `__init__` function	overriding of the `start` function
Agent Destruction	overriding of the `takeDown` function	-	-	overriding of the `end` function
Agent Execution	one thread per agent	synchrone execution	synchrone execution, `__step__` function calls	one thread per agent and synchrone execution, both need overriding of the `live` function
Hierarchical Multiagent System	None	None	Hierarchy of agents	Holarchy

Table 1: Mapping of the SARL concepts to the existing agent platforms

All the existing platforms that are included in this comparison provide a message-based interaction mechanism: agents are sending messages to other agents. The receiving agents decide to get the messages from their mail boxes. This approach corresponds to a pro-active behavior: consuming a message is a direct and explicit decision of the agent. The messages are not given to the agent as part of its reactive behavior. It does not fit the reactive interaction behavior, based on events, that is expected by the SARL developers. The creation of an SRE, based on all the above frameworks needs a complete recast of the data exchange mechanism.

The agent's life-cycle is almost supported in the same way by all the above platforms. Initialization and destruction functions are invoked at the beginning and end of the agent's life. In between, the agent's life is supported by a specific function that is called regularly by the agent framework. SARL specification does not assume the existence of this function. In order to make each platform compliant with SARL, this agent running function should read the agent mailbox and fires corresponding events, for example.

Agent execution is another key feature of an agent framework. The existing frameworks use a single thread to each agent. According to the execution mechanism that is expected by SARL, the execution mechanism should be adapted to enable threaded execution of the event handlers. Or, this parallel execution should be simulated in order to give the illusion to the agent's code that the event handlers are executed in parallel. Regarding the agent execution function mentioned above, it considered as the place where to specify the execution mechanism mapping.

The last criterion focuses on the support of hierarchical systems. Two agent frameworks support explicitly such systems, i.e. GAMA and Janus. Only Janus fully supports the concept of holon.

According to all these assessments, we have decided to adapt the Janus platform in order to create a run-time environment for the SARL applications. Indeed, Janus is the framework, which covers the largest amount of features that are expected by SARL developers. The key element that has been considered for taking this decision is the support of holonic multi-agent systems, which is the best within Janus, from our point of view. This decision is also taken according to the software development effort to be spent for creating a SRE: lower is the amount of code to be written, higher is our interest to the SRE candidate.

6. The Janus Platform as a SARL Run-time Environment

The Janus platform[7] is re-designed and re-implemented in order to serve as the software execution environment for the SARL programs. This revised version of Janus implements all the required infrastructure to execute a MAS, which is programmed using SARL. It fulfills the SARL requirements, such as fully distributed execution of the agent's behaviors, and the automatic discovery

[7] Official website: `http://www.janusproject.io`

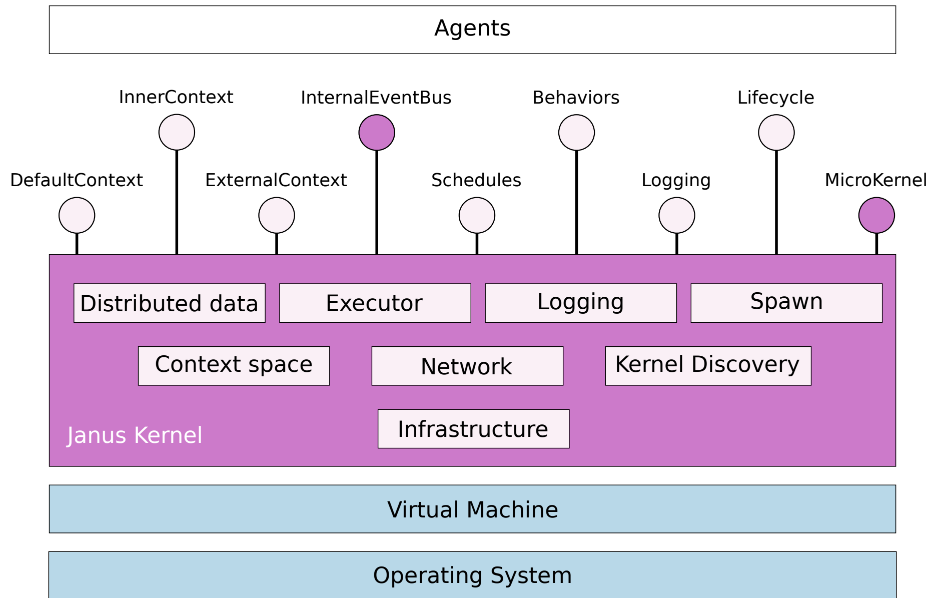

Figure 4: Architecture of the Janus platform

of kernels. Janus adopts best practices in current software development, such as the Inversion of Control[8]. It also benefits from new technologies like Distributed Data Structures[9]. The main purpose of this work is to adapt Janus to become a SRE, and therefore provides an implementation to each of the built-in capacities.

6.1. General Service-based Architecture

A service-oriented architecture (SOA) is a style of software design where services are provided to the other components by application components. The basic principles of SOA are independent of vendors, products and technologies [34]. A service is a discrete unit of features that can be accessed remotely, acted upon, and updated independently, such as retrieving a credit card statement on-line.

A service has four properties according the definition given by the Open Group[10]: (i) it logically represents a business activity with a specified outcome; (ii) it is self-contained; (iii) it is a black box for its consumers; and (iv) it may consist of other underlying services. Different services can be used in conjunction to provide the functionality of a large software application [35]. So far, the definition could be a definition of modular programming in the 1970s.

[8]Google Guice: `https://code.google.com/p/google-guice/`
[9]In-Memory Data Grid like Hazelcast: `http://www.hazelcast.com`
[10]Service-Oriented Architecture Standards: `http://www.opengroup.org/standards/soa`

Service-oriented architecture is less about how to modularize an application. It is more about how to compose an application by integration of distributed, separately-maintained and deployed software components. SOA is enabled by technologies and standards that make it easier for components to communicate and cooperate.

According to these principles, the Janus platform is redesigned in order to provide eight services. As illustrated by Figure 4, these services constitute the basis of the built-in capacity implementation. The Janus services are aggregated into two categories:

1. infrastructure services, which are managing the relationship with the underlying operation system and the virtual machine.

2. services that are managing features that are accessible to the built-in capacities.

These services are detailed in the following sections according to the given presentation template:

Description:	Goal and activities that are carried by this service out.
Start-up:	Description of the activities that are executed at the start-up of the service.
Shut-down:	Description of the activities that are executed at the shut-down of the service.
Functions:	Descriptions of the functions that are provided by the service in its public interface.
Events:	List of the events that are fired by the service, and that could be listened by external objects.

The rest of this section provides details on the height services defined within the Janus framework.

6.1.1. Infrastructure Service

Description:	Manages the resources that are provided by the Java Virtual Machine or the operating system.
Start-up:	Start up the Hazelcast manager (`http://www.hazelcast.com`). Hazelcast is an in-memory data grid library that is used to create and manage data structures distributed other a computer network.
Shut-down:	Shut down the Hazelcast library.
Functions:	-
Events:	-

6.1.2. Logging Service

Description:	Enables an agent to output messages in a specific log, with associated emergency level. The service implementation is based upon the Oracle Logging API, included into all the Java run-time environments.
Start-up:	-
Shut-down:	-
Functions:	`debug`, `info`, `warning` and `error` functions. They take the message to be logged as argument.
Events:	-

6.1.3. Executor Service

Description:	This service provides the functions for launching tasks in parallel. The implementation is based on the Java `Executors` utility class, which provides tools for launching single-run and periodic tasks.
Start-up:	Create and initialize a Java executor service for single-run tasks, and one for periodic tasks.
Shut-down:	Stop the Java executor services.
Functions:	• `execute(r)`: execute the task r in parallel. • `executeMultipleTimesInParallelAndWaitFor–Termination(r, n)`: execute n instances of the task in parallel. • `schedule(t, r)`: execute the task r in t milliseconds. • `scheduleAtFixedRate(t, r)`: execute the task r every t milli-seconds. • `scheduleAtFixedDelay(t, r)`: execute the task r infinitely, and wait t milliseconds between each run.
Events:	-

6.1.4. Context-Space Service

Description:	This service is in charge of maintaining the repositories of the agent contexts and the agent interaction spaces that are created in the system. This service is also in charge of routing the events between agents through the spaces. The agent execution unit in Janus is the event handler: the part of the SARL agent that is executed when a specific event is received. Each of these units are executed in parallel to the other units, even within the same agent.
Start-up:	Synchronization of the context and space repositories with other Janus instances over the computer network.
Shut-down:	-
Functions:	• `createContext(id)`: create a context with the given identifier. • `removeContext(id)`: remove the context with the given identifier, and destroy all the spaces inside the context. • `getContexts()`: return all the existing contexts. • `getContext(id)`: return the context with the given identifier. • `createSpace(c, id)`: create a space in the context c with the given identifier. • `removeSpace(c, id)`: remove the space with the given identifier from the context c. • `getSpaces(c)`: return all the existing spaces in the context c. • `getSpace(c, id)`: reply the context with the given identifier in the context c.
Events:	• `ContextCreated`, `ContextDestroyed` when a context was created or destroyed. • `SpaceCreated`, `SpaceDestroyed` when a space was created or destroyed.

Janus enables the distribution a SARL program over a network of (virtual) machines — (V)M's. From the SARL agent point of view, this distribution over (V)M's is hidden. Indeed, we argue that any SARL program runs on a big virtual machine that is covering all the connected (V)M's. These low-level machines are not visible to the SARL agent. Consequently, a context is seen as a single entity over the (V)M's by the agents. From the Janus point of

view, each context instance is a shared object over the (V)M's. Thanks to the underlying Hazelcast in-memory data grid library, the different context instances are replicated and dynamically synchronized over the (V)M's. Spaces are also shared upon the same infrastructure.

A safety question arises when enabling (V)M communication: what happens when two (V)M's cannot communicate any more? Thanks to the Hazelcast library, no data is removed from the shared data structures (context list, space list, agent list, etc.) It means that the agents can continue to send events to agents that are at the other side of the lost connection. But, Janus does not send the events to the remote agent because of the connection loss. According to the Janus specification, there is no warranty that an event is delivered to another agent into a remote (V)M. Consequently, the agent communication protocol should take care of any loss of event. When the (V)M's connection is back, Hazelcast library synchronizes the local instances of the data structures with the remote instances.

6.1.5. Spawning Service

Description:	This service is in charge of managing the agent's life-cycle. It creates instances of agents, and registers them to the other services of the framework. This service provides the functions for stopping the agents. An agent can be killed only if it has no member agent inside (see Section 2.6). The service ensures that the agent's life-cycle events are fired to the agent: `Initialize` and `Destroy` events.
Start-up:	-
Shut-down:	Stop all the running agents.
Functions:	• `spawn(t, n)`: create `n` instances of agent of type `t`. • `killAgent(id)`: destroy the agent when the given identifier.
Events:	• `AgentSpawned` when an agent was created. • `AgentDestroy` when an agent was destroyed.

6.1.6. Networking Services

Three networking services are provided by the Janus platform: kernel discovery, distributed data, and network event routing services.

Name:	Kernel discovery
Description:	The kernel discovery service is in charge of maintaining a up-to-date list of the Janus kernels that are alive on a local computer network. This service uses the Hazelcast library, which is already maintaining a list of the Hazelcast nodes over the network.
Start-up:	Advertise the current kernel over the computer network.
Shut-down:	Notification of the disappearance of the current kernel to the other Janus kernels.
Functions:	• `getKernels`(): return the list of the Janus kernels.
Events:	• `KernelDiscovered` when a Janus kernel was detected. • `KernelDisappeared` when a Janus kernel is no more reachable.

Name:	Distributed data structures
Description:	The distributed data service provides functions to create data structures (hash tables, lists, etc.) that are accessible and synchronized over the computer network. This service uses the Hazelcast library.
Start-up:	-
Shut-down:	-
Functions:	`newMap`(`id`) and `newList`(`id`) are provided to create the data structures with the given identifiers. These functions replicate the data structures in all the kernel thanks to the Hazelcast library.
Events:	-

Name:	Network event routing
Description:	The network event routing service is in charge of routing the events that are fired by the agents to the agents that are hosted on a remote computer. This service opens a socket channel, based on the ZeroMQ library[11] to each remote Janus kernel. This channel will be used for sending events to the remotely hosted agents.
Start-up:	-
Shut-down:	-
Functions:	-
Events:	• `EventReceived` when an event is received from a remote Janus kernel.

6.2. Built-in Capacity Implementation

SRE creates and injects the BICs in an agent before its execution begins. Janus platform provides an implementation for each BIC described in Section 4.2. Table 2 provides the mapping from a BIC to the Janus services that are used for its implementation. Indeed, the BICs call the Janus services in order to realize there behaviors. The concrete implementation code may be found on Github: `https://github.com/sarl/sarl/tree/master/sre/io.janusproject/io.janusproject.plugin`.

Built-in Capacity	Janus Services
`ExternalContextAccess`	Context-Space
`InnerContextAccess`	Context-Space
`Behaviors`	Context-Space, Executor
`Lifecycle`	Spawning, Context-Space
`Schedules`	Executor
`DefaultContextInteractions`	Context-Space
`Logging`	Logging
`Time`	Executor

Table 2: Mapping between the Built-in Capacities and the Janus Services.

[11]ZeroMQ: `http://zeromq.org/`

7. Performance Evaluation

In order to be able to measure the performance of the new Janus implementation, we created a very simple stigmergy-inspired ping-pong application [17]. A central agent representing the agent environment, as defined by Weyns et al. [14] and Galland and Gaud [23], is introduced as the mean of communication between the other agents. Consequently, two agent types are considered: (i) the application agents, which are sending the ping-pong events, and (ii) the environment agent, which represents the environment in which the application agents are located. The times T_0, T_1, T_A, T_B, T_C and T_D mentioned below are simulated application time values.

In the time period starting at T_0, every application agent has 20 % probability to emit a *ping* message to X other agents where $X \sim \mathit{Uniform}(1 : 100)$. The message needs to be delivered in the time period $T_d \sim \mathit{Uniform}(T_1 : T_e)$ where T_e denotes the end of simulated time. The measured time is illustrated in Figure 5. Where T_0 and T_1 denote the start and the end of the interval. T_a denotes the end of the reception of the events sent by the environment agent to the application agents. T_b is the end of "application level payload work" done by the simulation agents. And finally, T_c is the end of delivering of the `AgentIsReadyEvent` events to the environment agent. For every time period, the amount of emitted messages is computed together with the total amount of time needed to execute this time period. Experiments are realized for 200 agents on a Linux Ubuntu 14.04LTS laptop with 8GB memory and a Intel Core i5-4210M CPU 2.60GHz × 4. The number of time periods that are simulated is 2 500.

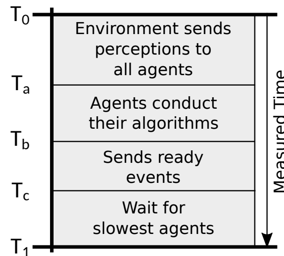

Figure 5: An overview of the time measurement in our experiments.

Experimental results are illustrated in the graph represented in Figure 6. In our experiments, all the application agents have the same actions to do. Consequently, they have approximately the same execution time. It is clear to see that the execution time follows a constant tendency, and hence seems to be independent of the number of processed events over the full range of observations. The execution time for a single period between two consecutive increments of simulated time includes: perception of the environment, application specific *payload* work and end-of-period notification. The duration required for the payload work

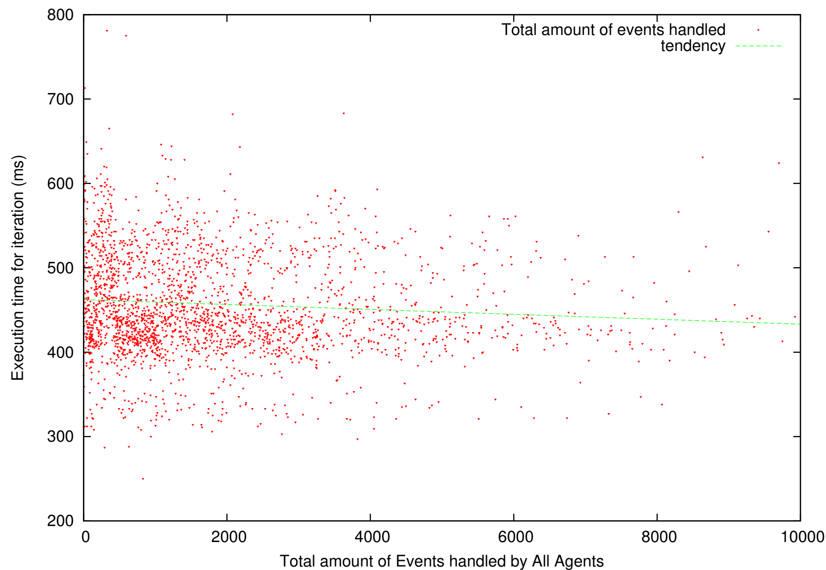

Figure 6: Graph that represents the total amount of events handled in a specific iteration (x-axis) against the total execution time for that iteration in ms (y-axis) for the case of 200 agents and 2 500 iterations [17].

in the experiment is negligible. The large variance of the execution time masks the expected dependency on the number of events.

8. Conclusion and Perspectives

SARL is a general-purpose agent-oriented programming language. This language aims at providing the fundamental abstractions for dealing with essential agent features: concurrency, distribution, interaction, decentralization, reactivity, autonomy and dynamic reconfiguration.

Every programming language specifies an execution model, and many implement at least part of that model in a runtime system. In the case of SARL programs, the SARL run-time environment (SRE) provides the tools and the features that are mandatory for running such a program. In this paper, we present the adaptation of the Janus platform for becoming the official and default SRE. Janus adopts the best practices in current software development, such as Inversion of Control, and benefits from new technologies like Distributed Data Structures.

The major perspectives of this work are listed below. First, Janus platform is a standard Java application for which the performances must be analyzed in detail, in order to be optimized accordingly. A comprehensive and systematic comparison of the existing agent frameworks and the new version of Janus will be realized.

Migration of agents over the different Janus kernels is not yet supported. A specific service will be added into Janus, and a built-in capacity provided within the SARL API, in order to query to migrate[12].

Other agent-based platform may serve as SRE. GAMA platform [6] is a possible candidate for creating a simulation environment for spatial and geographic applications. Gazebo platform [36, 37] is another candidate for creating simulators of robots, including drones.

Finally, most of the embedded systems cannot execute a Java application. The need of a specific SARL compiler, which generates C/C++ program arises. A perspective of this work is to extend the SARL compiler for embedded systems, cloud platforms, and computer clusters, which are specific run-time environments.

Acknowledgements

We would like to thank Glenn Cich and Luk Knapen, who have highly contributed to the performance evaluation of the new Janus platform [17].

References

[1] M. Cossentino, S. Galland, N. Gaud, V. Hilaire, A. Koukam, How to control emergence of behaviours in a holarchy, in: the Int. Workshop on Self-Adaptation for Robustness and Cooperation in Holonic Multi-Agent Systems (SARC-2008) at the Second International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008), IEEE Computer Society, Venice, Italy, 2008.

[2] S. A. DeLoach, The MaSE Methodology, Springer US, Boston, MA, 107–125, 2004.

[3] M. Wooldridge, N. R. Jennings, D. Kinny, The Gaia Methodology for Agent-Oriented Analysis and Design, Autonomous Agents and Multi-Agent Systems 3 (3) (2000) 285–312.

[4] F. L. Bellifemine, G. Caire, D. Greenwoord, Developing Multi-Agent Systems with JADE, John Wiley & Sons, 2007.

[5] U. Wilensky, NetLogo, Tech. Rep., Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 1999.

[6] A. Grignard, P. Taillandier, B. Gaudou, D. Vo, N. Huynh, A. Drogoul, GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation, in: G. Boella, E. Elkind, B. Savarimuthu, F. Dignum, M. Purvis (Eds.), PRIMA 2013: Principles and Practice of Multi-Agent Systems, vol. 8291 of *LNCS*, Springer Berlin Heidelberg, 117–131, 2013.

[12] Agent migration feature: `https://github.com/sarl/sarl/issues/747`.

[7] S. Galland, N. Gaud, S. Rodriguez, V. Hilaire, Janus: Another Yet General-Purpose Multiagent Platform, in: the 7th Agent-Oriented Software Engineering Technical Forum (TFGAOSE-10), Agent Technical Fora, Agent Technical Fora, Paris, France, 2010.

[8] D. Glake, J. Weyl, C. Dohmen, C. Hüning, T. Clemen, Modeling Through Model Transformation with MARS 2.0, in: International Springer Simulation Conference, dOI: 10.22360/springsim.2017.ads.005, 2017.

[9] R. H. Bordini, J. F. Hübner, M. Wooldridge, Programming Multi-Agent Systems in AgentSpeak using Jason, Wiley, 1st edn., ISBN 978-0-470-02900-8, 2007.

[10] S. Rodriguez, N. Gaud, S. Galland, SARL: A General-Purpose Agent-Oriented Programming Language, in: Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on, vol. 3, 103–110, 2014.

[11] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Pearson Education, Inc, 2nd edn., ISBN 0-201-10088-6, 2006.

[12] S. Rodriguez, N. Gaud, V. Hilaire, S. Galland, A. Koukam, An analysis and design concept for self-organization in Holonic Multi-Agent Systems, in: the International Workshop on Engineering Self-Organizing Applications (ESOA'06), Springer-Verlag, 62–75, 2006.

[13] A. Ricci, M. Viroli, A. Omicini, Programming MAS with Artifacts, in: International Workshop on Programming Multi-Agent Systems, Springer Verlag, 2005.

[14] D. Weyns, A. Omicini, J. Odell, Environment as a First-class Abstraction in Multi-Agent Systems, Autonomous Agents and Multi-Agent Systems 14 (1) (2007) 5–30, ISSN 1387-2532.

[15] S. Galland, F. Balbo, N. Gaud, S. Rodriguez, G. Picard, O. Boissier, A multidimensional environment implementation for enhancing agent interaction, in: R. Bordini, E. Elkind (Eds.), 14th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS15), IFAAMAS, ACM In-Cooperation, Istanbul, Turkey, ISBN 978-1-4503-3413-6, 1801–1802, URL http://www.aamas2015.com/en/AAMAS_2015_USB/aamas/p1801.pdf, 2015.

[16] S. Rodriguez, S. Galland, N. Gaud, A New Perspective on Multi-Agent Environment with SARL, in: International Workshop on Communication for Humans, Agents, Robots, Machines and Sensors, Procedia Computer Science, Elsevier, Belfort, France, ISSN 1877-0509, 526–531, URL http://www.sciencedirect.com/science/article/pii/S1877050915017275, best Paper Award, 2015.

[17] G. Cich, S. Galland, L. Knapen, A.-U.-H. Yasar, T. Bellemans, D. Janssens, Addressing the Challenges of Conservative Event Synchronization for the SARL Agent-Programming Language, in: the 15th International Conference on Practical Applications of Agents and Multi-Agent Systems, Springer, 2017.

[18] D. Lizondo, P. Araujo, A. Will, S. Rodriguez, Multiagent Model for Distributed Peak Shaving System with Demand-Side Management Approach, in: 2017 First IEEE International Conference on Robotic Computing (IRC), 352–357, dOI: 10.1109/irc.2017.50, 2017.

[19] M. Feraud, S. Galland, First Comparison of SARL to Other Agent-Programming Languages and Frameworks, in: International Workshop on Agent-based Modeling and Applications with SARL (SARL 2017), Procedia Computer Science, Elsevier, dOI: 10.1016/j.procs.2017.05.389, 2017.

[20] S. Galland, F. Balbo, G. Picard, O. Boissier, N. Gaud, S. Rodriguez, Environnement multidimensionnel pour contextualiser les interactions des agents. Application à la simulation du trafic routier urbain., Special Issue on Multiagent Systems of the "Revue d'Intelligence Artificielle" 30 (1-2) (2016) 81–108.

[21] G. Cich, L. Knapen, S. Galland, J. Vuurstaek, A. Neven, T. Bellemans, Towards an Agent-based Model for Demand-Responsive Transport Serving Thin Flows, in: The 5th International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS 2016), Procedia Computer Science, Elsevier, 2016.

[22] G. Basso, M. Cossentino, V. Hilaire, F. Lauri, S. Rodriguez, V. Seidita, Engineering multi-agent systems using feedback loops and holarchies, Engineering Applications of Artificial Intelligence 55 (2016) 14 – 25, ISSN 0952-1976, URL http://www.sciencedirect.com/science/article/pii/S0952197616300999, dOI: 10.1016/j.engappai.2016.05.009.

[23] S. Galland, N. Gaud, Organizational and Holonic Modelling of a Simulated and Synthetic Spatial Environment, E4MAS 2014 - 10 years later, LNCS 9068 (1) (2015) 1–23, URL http://www.springer.com/us/book/9783319238494.

[24] S. Galland, F. Balbo, N. Gaud, S. Rodriguez, G. Picard, O. Boissier, Contextualize Agent Interactions by Combining Social and Physical Dimensions in the Environment, in: Y. Demazeau, K. Decker, F. De la prieta, J. Bajo perez (Eds.), Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection. Lecture Notes in Computer Science 9086., Springer International Publishing, 107–119, dOI: 10.1007/978-3-319-18944-4˙9, 2015.

[25] A. Koestler, The Ghost in the Machine, Hutchinson, 1967.

[26] K. Wilber, Sex, Ecology, Spirituality: The Spirit of Evolution, Shambhala, ISBN 9781570627446, 2000.

[27] E. van Leeuwen, D. Norrie, Holons and holarchies [intelligent manufacturing systems], Manufacturing Engineer 76 (2) (1997) 86–88.

[28] C. Gerber, J. Siekmann, G. Vierke, Holonic Multi-Agent Systems, Tech. Rep. DFKI-RR-99-03, Deutsches Forschungszentrum für Künztliche Inteligenz - GmbH, Postfach 20 80, 67608 Kaiserslautern, FRG, 1999.

[29] J. Ferber, Multi-agent systems: an introduction to distributed artificial intelligence, Addison-Wesley, 1999.

[30] J. H. Holland, Hidden order: how adaptation builds complexity, Addison-Wesley, Reading, Mass., 1995.

[31] J. Odell, M. Nodine, R. Levy, A Metamodel for Agents, Roles, and Groups, in: J. Odell, P. Giorgini, J. Müller (Eds.), Agent-Oriented Software Engineering V, no. 3382 in LNCS, Springer Berlin Heidelberg, 78–92, 2005.

[32] M. Cossentino, N. Gaud, V. Hilaire, S. Galland, A. Koukam, ASPECS: an agent-oriented software process for engineering complex systems - How to design agent societies under a holonic perspective, Autonomous Agents and Multi-Agent Systems 2 (2) (2010) 260–304.

[33] D. Case, S. DeLoach, Applying an O-MaSE Compliant Process to Develop a Holonic Multiagent System for the Evaluation of Intelligent Power Distribution Systems, in: M. Cossentino, A. El Fallah Seghrouchni, M. Winikoff (Eds.), Engineering Multi-Agent Systems, no. 8245 in LNCS, Springer Berlin Heidelberg, 78–96, 2013.

[34] Service-Oriented Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-38284-3, 89–113, URL `http://dx.doi.org/10.1007/978-3-540-38284-3_5`, 2007.

[35] A. T. Velte, Cloud Computing: A Practical Approach, McGraw Hill, 2010.

[36] C. Aguero, N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, B. Gerkey, S. Paepcke, J. Rivero, J. Manzo, E. Krotkov, G. Pratt, Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response, Automation Science and Engineering, IEEE Transactions on 12 (2) (2015) 494–506, ISSN 1545-5955.

[37] N. Koenig, A. Howard, Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, 2149–2154, 2004.

Stéphane Galland supports a PhD thesis in 2001 at the High National School of Mines of Saint-Etienne, France. He proposed a methodological approach for the design and the implementation of agent-based simulation of distributed industrial systems. In 2002, he integrated the Computer Science department of the Belfort-Montbéliard University of Technology, France, where he continues his research tasks on the topic of agent-based modeling and simulation of complex systems with a large scale and a multiview perspectives. In 2004 and 2005, Stéphane Galland is responsible of the courses of the doctoral school for his University. From 2007 to 2008, he is responsible of the courses of the specialty "Image, Interaction and Virtual Reality" of the Computer Science department. In 2013, Stéphane Galland obtains a French Accreditation to Supervise Research with the title "Methodology and tools for the agent-based simulation in virtual worlds." Stéphane Galland is one of the authors to the ASPECS methodology, the SARL agent-programming language, and the Janus agent platform. Since 2016, Stéphane Galland is the French Head of the ARFITEC exchange program named "Energy, Transport, Industry, Challenges for Tomorrow." In January 2017, Stéphane Galland integrates the Electronic, Computer Science and Imagery Laboratory (LE2I) as the Head of the Research team on "Intelligent Environments."

Sebastian Rodriguez is a Full Professor at the Department of Computer Science, Universidad Tecnológica Nacional (UTN), Argentina. He is also the founder and Head of the Advanced Informatics Technology Research Group (GITIA) and an associate researcher of the Systems and Transportation Laboratory at the University of Technology of Belfort-Montbéliard (UTBM), France. He received a Computer Engineer degree from Universidad Nacional de Tucumán, a M.Sc. degree in computer science from the University of Franche-Comté and a Ph.D. degree in computer science of the UTBM.

Nicolas Gaud received his PHD in Computer Science from the University of Technology of Belfort-Montébliard (UTBM) in 2007. In 2005, he received his engineering degree in computer science from the UTBM and a MSc in Computer Science, Automatic and Manufacturing Systems from the University of France-Comté (UFC). He is now Associate Professor at the UTBM and full researcher at the Systems and Transport Laboratory of the research institute on Transport, Energy, and Society (IRTES-SeT), he is also an external member of the GITIA. His main research interests deal with the modeling, analysis and simulation of complex systems using Agent-Oriented Software Engineering (AOSE), Holonic Multiagent Systems and Multiagent-based simulation. He is also involved in various industrial projects dealing with the simulation of virtual entities (pedestrian, transporta-

tion systems, etc) in virtual environments.