# SARL: a general-purpose agent-oriented programming language

Sebastian RODRIGUEZ
GITIA – UTN,
4000 Tucumán, Argentina
http://www.gitia.org
email: sebastian.rodriguez@gitia.org

Nicolas GAUD
IRTES-SeT – UTBM,
Rue Ernest Thierry-Mieg,
90010 Belfort cedex, France.
http://multiagent.fr
email: nicolas.gaud@utbm.fr

Stéphane GALLAND
IRTES-SeT – UTBM,
Rue Ernest Thierry-Mieg,
90010 Belfort cedex, France.
http://multiagent.fr
email: stephane.galland@utbm.fr

***Abstract*—Complex software systems development require appropriate high-level features to better and easily tackle the new requirements in terms of interactions, concurrency and distribution. This requires a paradigm change in software engineering and corresponding programming languages. We are convinced that agent-oriented programming may be the support for this change by focusing on a small corpus of commonly accepted concepts and the corresponding programming language in line with the current developers' programming practices. This papers introduces SARL, a new general-purpose agent-oriented programming language undertaking this challenge. SARL comes with its full support in the Eclipse IDE for compilation and debugging, and a new version 2.0 of the Janus platform for execution purposes. The main perspective that guided the creation of SARL is the establishment of an open and easily extensible language. Our expectation is to provide the community with a common forum in terms of a first working testbed to study and compare various programming alternatives and associated metamodels.**

***Index Terms*—Agent-oriented Programming, Programming Language, Holonic multiagent system, Recursive agents**

## I. Introduction

This paper introduces a new general-purpose agent-oriented programming language (APL) called SARL[1]. This language aims at providing the fundamental abstractions for dealing with concurrency, distribution, interaction, decentralization, reactivity, autonomy and dynamic reconfiguration. These high-level features are now considered as the major requirements for an easy and practical implementation of modern complex software applications. We are convinced that the agent-oriented paradigm (AOP) holds the keys to effectively meet this challenge.

Indeed, the ultimate goal of agent-oriented software engineering (AOSE) and programming is the establishment of a new standard paradigm able to succeed object-oriented paradigm to ease the development of complex software applications by providing higher-level abstractions. However, it is clear that AOP has not yet a significant influence on the current mainstreams and standards in software programming. One possible explanation may be that most of contributions in AOP remain highly theoretical and rather far from industrial concerns and related developers' programming practices. We adopt a different perspective capitalizing on previous experiences aspiring to define a small corpus of concepts only focusing on the key principles. Considering the variety of existing approaches and meta-models, but also the lack of established standards in the field of AOSE and more generally multi-agent systems, our approach remains as generic as possible and highly extensible to easily integrate new concepts and features. The idea is not to respond to every issues immediately but rather to provide a first set of concepts, the corresponding programming language as well as a set of tools for supporting its implementation. Our expectations is to provide the community with a common forum in terms of a first working testbed to study and compare various programming alternatives and associated metamodels.

In this perspective, SARL is platform-independent and agent's architecture-agnostic. SARL provides a set of agent-oriented first-class abstractions directly at the language level, but it was designed to ease the integration and the mapping of concepts provided by other metamodels. SARL itself exploits this extension mechanism for defining its own extensions (organizational, event-driven, etc.). SARL also natively supports the notion of holonic multiagent systems and recursive agent (also called Holon), but does not force the developer to use it.

SARL comes with its full support in the Eclipse IDE for compilation and debugging purposes. We also provide a first set of tools to support its execution based on the new version 2.0 of the Janus[2] platform, but it can be linked with other existing agent platforms and frameworks. SARL remains completely independent from Janus. SARL compiler generates Java code that can then be integrated within any agent platforms supporting Java librairies. Janus is one of them. This new version of Janus integrates and thus benefits from the last advances and new patterns of Object-oriented programming like Inversion of Control, event-driven communication, distributed objects, etc.

The remainder of this article is organized as follows. Section II provides an early glimpse of the SARL language and its main characteristics. The SARL metamodel and its main

[1] http://www.sarl.io

[2] http://www.janusproject.io

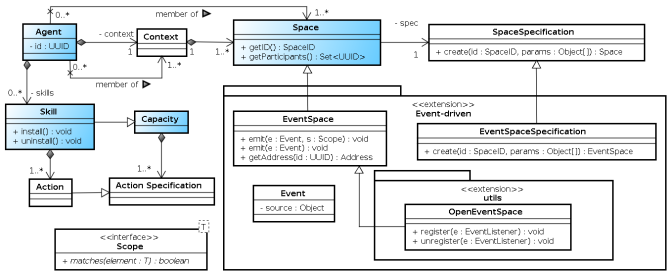


Fig. 1. SARL Main Concepts

concepts are then detailed in the following three sections, each one focusing on one of the fundamental dimensions of a multiagent system. Section III concentrates on the individual dimension and the agent abstraction. Section IV addresses the notion of Interaction. Finally, Section V shows how SARL provides the developer with the means to exploit the power of the holonic perspective. Section VI briefly describes the new version 2.0 of Janus as a VMs for SARL. Section VII discusses the related works on AOP, before concluding this paper by presenting the future works in Section VIII.

## II. Overview

The main perspective guiding 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 metamodel of SARL is based on four main concepts: Agent, Space, Capacity and Skill. The core metamodel of SARL is presented in Figure 1 and the main concepts are colored in light blue. Each of these concepts will be detailed in the following sections as well as the corresponding piece of SARL code to illustrate their practical use. In this paper, the key definitions of SARL concepts are presented in bold.

Basically, **a Multiagent System (MAS) in SARL is programmed as a collection of Agents interacting together in a collection of shared distributed Spaces**. Each agent has a collection of Capacities describing what it is able to perform, its personal competences [1, 2]. Each Capacity may then be realized/implemented by various Skills. We can draw the parallel with the concepts of Interfaces and corresponding implementation classes in object-oriented languages. To implement specific architectures (like BDI, reasoning, reactive, hybrid, etc.) developers should develop their own capacities and corresponding skills providing the agents with new exploitable features.

Despite its open nature, SARL imposes some fundamental principles to be respected by the various VMs that wants to support it. First of all, the implementation of Space must be fully distributed and the execution layer must be abstracted from agents. SARL encourages a massively parallel execution of Agents and Behaviors. SARL is fully interoperable with Java to easily reuse all the contributions provided by the Java community, but also to facilitate the integration and evolution of legacy systems. One of the key principles governing SARL consists in not imposing a predefined way for Agents to interact within a Space. Similarly, the way to identify agents is dependent on the type of Space considered. This allows to define different types of interaction mechanisms and models on Spaces, more details on this point in Section IV-B.

Strongly inspired by new languages like Scala[3] and Clojure[4] but also Ruby, we try to implement in SARL an intuitive syntax with a shallow learning curve. SARL has been developed on top of Xtext[5] that enables to easily build your own domain-specific language. We thus directly benefits from a working parser and linker as well as a direct integration in Eclipse. The complete definition of the SARL's grammar in Xtext format is available on GitHub[6]. Based on Xtext, itself using ANTLR*, SARL grammar is a LL* deterministic context-free grammar.

The full set of source codes presented in this article is freely available at https://github.com/sarl/sarl-demos, as well as a collection of simple demos demonstrating the various features of SARL. A very simple example of a multiagent application will be used throughout this article to demonstrate the possibilities offered by the SARL language and illustrate its syntax. This application involves two agents: a first agent called `FactorialQueryAgent` who asks a second agent `FactorialAgent` to calculate the factorial of a given integer and waits for the result. Once it receives the result, it displays it and both agents die.

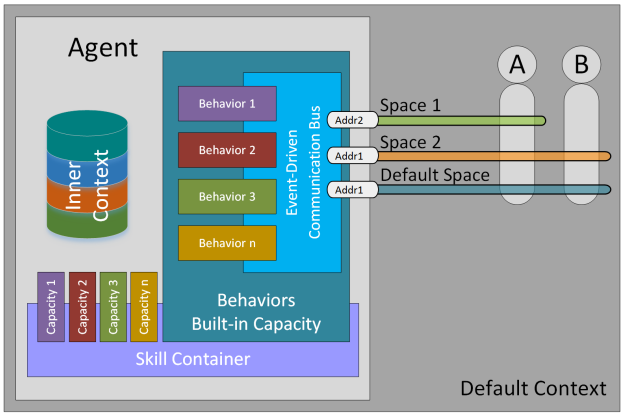


Fig. 2. An Agent in SARL

## III. The Agent abstraction, the individual dimension

### A. Agent and Behavior

**An agent is an autonomous entity having a set of skills to realize the capacities it exhibits**. An agent has a set of built-in capacities considered essential to respect the commonly accepted competences of agents, such autonomy, reactivity,

[3]http://www.scala-lang.org
[4]http://clojure.org
[5]https://www.eclipse.org/Xtext/
[6]https://raw.github.com/sarl/sarl/master/lang/plugins/io.sarl.lang/src/io/sarl/lang/SARL.xtext

proactivity and social capacities. Figure 2 describes the open architecture of an Agent in SARL. The full set of Built-In Capacities (BIC) will be presented in Section III-B. Among these BICs, is the *Behaviors* capacity that enables agents to incorporate a collection of behaviors that will determine its global conduct. An agent has also a default behavior directly described within its definition. **A Behavior maps a collection of perceptions represented by Events to a sequence of Actions.** The various behaviors of an agent communicates using an event-driven approach. **An Event is the specification of some occurrence in a Space that may potentially trigger effects by a listener** (e.g. agent, behavior, etc.).

For clarity reasons, let's describes the definition of Agent using our example. Listing 1 presents the definition of the `FactorialAgent` and the various events governing its overall behavior. This agent will wait for other agent's request to calculate (`Calculate` event) a factorial. Once computed it will notify the result using the `ComputationDone` event.

```
1  package io.sarl.demos.queryfactorial
2  /* Import section */
3  event Factorial {
4      var number : Integer
5      var value : Integer
6  }
7  event Calculate {
8      var number : Integer
9  }
10 event ComputationDone {
11     var result : Integer
12 }
13 agent FactorialAgent {
14     uses Lifecycle, Behaviors, Logging,
           DefaultContextInteractions
15     var upto : Integer = 10
16     on Initialize {
17         setSkill(Logging,new BasicConsoleLogging(this))
18         info("Factorial initialized")
19     }
20     on Factorial [occurrence.number < upto ] {
21         wake(new Factorial => [ number = occurrence.
               number.increment ; value = occurrence.value
                * (occurrence.number.increment) ])
22     }
23     on Factorial [ occurrence.number == upto ] {
24         info("Factorial of " + upto + " is " +
               occurrence.value)
25         emit(new ComputationDone=>[result=occurrence.
               value])
26         killMe
27     }
28     on Calculate {
29         this.upto = new Integer(occurrence.number);
30         info("Received Calculate for " + this.upto)
31         wake(new Factorial => [ number = 0 ; value = 1
               ])
32     }
33     on Destroy {
34         debug("Factorial: I'm ready to die")
35     }
36     def increment(nb:Integer) : Integer {
37         nb+1
38     }
39 }
```

Example Code 1. Event and Agent in SARL

An agent is declared with the `agent` keyword (line 13). In the agent's body block, we can declare Mental States (in the form of attributes), Actions and Behaviors.

The actions an agent can perform can be specified by Capacities or natively inside the agent body. For instance, in line 36 a native action `increment` is defined. The keyword `uses` imports the actions defined in Capacities so that they can be accessed directly as an agent native action.

The `FactorialAgent` declares an attribute called `upto` using the `var` keyword (line 15). This attribute will be used by the behaviors to know for which number the agent is calculating the factorial for. It is important to notice that the type of `upto` is actually `java.lang.Integer` and that we can then directly use any Java class.

The agent must then declare its perceptions and the sequence of actions it wants to perform for each perception. This is achieved using the clause `on <Perception> [<guard>] {<body>}` The `FactorialAgent` declares five behaviors (lines 16, 20, 23, 28, 33).

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 in line 7. Events can carry information, in our case the number we want the factorial for.

When the agent is spawned and ready for execution, it will receive an `Initialize` event (see III-A). This allows the agent to install new Capacities, access resources, etc. A detailed description of Capacities and Skills can be found in Section III-B.

When the `Calculate` event is perceived (line 28), the agent can access the event's instance using the `occurrence` keyword. On line 29 it will set the `upto` attribute using the information for the `Calculate` event occurrence. The `info` action is imported from the `Logging` Capacity and simply prints a string to the console.

Now the agent should start 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). To calculate the factorial it will fire an internal event of type `Factorial` using the `wake` action.

Two behaviors are declared for `Factorial` (line 20 and 23). 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 on line 20 will only be executed if `occurrence.number < upto` evaluates to `true`. This behavior simply calculates the factorial for the next integer and fires a `Factorial` event again. It is important to notice that actions can be called using the first argument as callee. Therefore one can use `increment(1)` or `1.increment` indistinctly. This syntax sugar can provide fluent chain of actions[7].

Likewise, when the factorial for the requested number (stored in `upto` attribute) is found the behavior on line 23 will be executed. The `emit` action fires an event in the *Default Space* of the *Default Context* to notify that the Computation is finished. Details on agent interaction are provided in Section IV-B. After that the agent stops its execution using the `killMe` action from the *Lifecycle* capacity.

[7]This feature comes as a direct result of SARL being based on Xtext and Xbase.

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 (more details on this aspect in Section V-2). While `emit` action enables to fire an event in a given space.

*Agent's Lifecycle:* SARL does not imposes a specific agent's control loop. Indeed, when agents are spawned (by the host VM or other agent), the VM is in charge of creating the agent instance and installing the skills associated to the built-in capacities into the agent. Then, when the agent is ready to begin its execution, it fires the `Initialize` event. This event contains any 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), the VM will fire the `Destroy` event to enable 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. The key characteristic of Agents is their autonomy and no other agent should be able to stop its execution without its consent. The designer is then free to implement any control or authority protocol for their own application scenario.

### B. Capacity and Skill

**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 is defined in terms of pre- and post-conditions. **A Capacity is the specification of a collection of actions**. This specification makes no assumptions about its implementation. It could be used to specify what an agent can do, what a behavior requires for its execution, or what a group of agents as a whole may achieve with its collective behavior. **A Skill is a possible implementation of a capacity fulfilling all the constraints of this specification**. An agent can dynamically evolve by learning/acquiring new Capacities, but it can also dynamically change the Skill associated to a given capacity [1, 2]. 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.

Listing 2 describes the SARL syntax for defining a basic capacity `Logging` with two actions `debug` and `info`. `BasicConsoleLogging` is one possible skill implementing this capacity. The action `setSkill` enables the dynamic installation of a new Skill within the considered agent (see line 25).

```
1  /* ... */
2  capacity Logging {
3      def debug(s: String)
4      def info(s: String)
5  }
6
7  skill BasicConsoleLogging implements Logging {
8      new (owner:Agent){
9          super(owner)
10     }
11
12     def debug(s: String) {
13         System.out.println("DEBUG:"+s)
14     }
15
16     def info(s: String) {
17         System.out.println("INFO:"+s)
18     }
19 }
20
21 agent FactorialQueryAgent {
22     uses Lifecycle, Behaviors, Logging,
           DefaultContextInteractions
23
24     on Initialize {
25         setSkill(Logging,new BasicConsoleLogging(this))
26         info("Query sent with number 5")
27         emit(new Calculate=>[number=5])
28     }
29
30     on ComputationDone {
31         info("Query: Result of Factorial 5 is "+
               occurrence.result)
32         killMe
33     }
34
35     on Destroy {
36         debug("Query: I'm ready to die")
37     }
38 }
```

Example Code 2. Capacity and Skill in SARL

*Built-in Capacities:* Every agent in SARL has a set of built-in capacities 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 on the SARL language but the skill implementing them are provided by the VM. The VM is responsible for creating them and injecting them on the agent before their execution begins. Therefore, when the agent receives the `Initialize` event they are already available.

Figure 3 presents the current six BICs available and the actions they provide along their signatures:

- **ExternalContextAccess** provides access to the contexts that the agent is a part of and actions required to join/leave new contexts (see Section IV).
- **InnerContextAccess** provides access to the *Inner Context* of the Agent. This is keystone for holonic agent implementation (see Section V).
- **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) as well as the `killMe` action to stop its own execution.
- **Schedules** enables the agent to schedule tasks for future or periodic execution.
- The **DefaultContextInteractions** is actually provided for convenience. It assumes that the action will be performed on the agent's *Default Context* and its *Default Space*. For instance, the `emit` action is a shortcut for

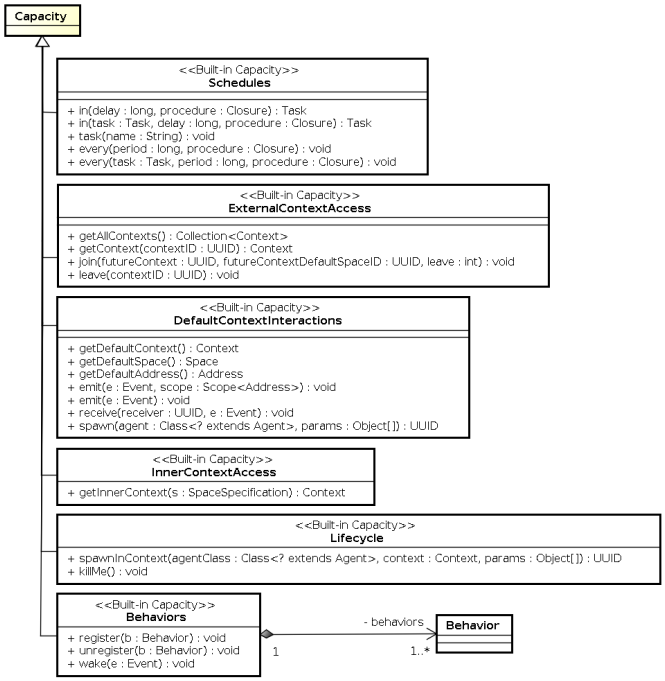


Fig. 3. Agent's Built-in Capacities

`defaultContext.defaultSpace.emit(...)`. Therefore, it is actually created on top of the other five BICs.

## IV. The Interaction Abstraction, the Collective Dimension

### A. Context and Spaces

**A Context defines the perimeter/boundary of a sub-system, and gathers a collection of Spaces**. In each context, there is at least one particular Space called *Default Space* to which all agents in this context 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. Since their creation, agents are incorporated into a context called the *Default Context* (see the upper part of Fig. 4, level n). For flat MAS, the concept of Context is relatively transparent to the developer, generally most applications will simply use the *Default Context*. The notion of Context makes complete sense when the developer is interested in hierarchical MAS and agents are considered composed or holonic (see Section V).

### B. Space and Space Specification

One of the key elements that characterize and differentiate the main multiagent approaches is how interactions between agents are described [3]. Some researchers focus on agent-to-agent interactions and corresponding protocols (e.g. FIPA ACL and protocols). Within organizational approaches, some consider the organization as a static partition of agents where agents interacts in groups through the roles they play (AGR, CRIO, etc.). Others focus on dynamic organizations and normative aspects (MOISE, OMNI, SODA, etc.). Another essential aspect of the interaction is the interaction Agent-Environment, especially in multiagent-based simulations. Each of these trends of multi-agent systems has led to numerous fruitful and innovative contributions. To remain generic, an APL should therefore not impose a single way of describing the interaction among agents, but rather attempt to provide means to implement each of these approaches. It is in this perspective that the concepts of Space and Space Specification were defined. **A Space is the support of the interaction between agents respecting the rules defined in a Space Specification. A Space Specification defines the rules (including action and perception) for interacting within a given set of Spaces respecting this specification.**

SARL natively defines a particular type of Space called *Event Space* to provide a support to event-driven interactions. Within an *Event Space*, agents communicate using Events, the BIC *DefaultContextInteractions* provides the agent with the means to emit and receive event, respectively using the `emit` actions and the `on` keyword in behavior definition. A *Default Space* is precisely an *Event Space*. Janus provides the full support for event spaces using a fully distributed approach with dynamic discovery of the participant agents all over the network.

Within an *Event Space*, the notion of Scope enables to precisely control/filter the potential recipients of an event. **A Scope is a predicate used to filter the potentially called listeners for a given event.** The most basic Scope is represented by a collection of Addresses (Agent, Role, etc.).

Listing 3 presents an example of such an event-driven communication. The `FactorialQueryAgent` emits a `Calculate` event in the *Default Space* of its *Default Context*. All agents register on this event will receive it. When the `FactorialAgent` receives this event, its starts the computation of the factorial, and when the result is available it forward it to the `FactorialQueryAgent` by emitting a `ComputationDone` event.

```
1  agent FactorialQueryAgent {
2  /* ... */
3      on Initialize {
4          setSkill(Logging,new BasicConsoleLogging(this));
5          info("Query sent with number 5")
6          emit(new Calculate=>[number=5])
7      }
8      on ComputationDone {
9          info("Query: Result of Factorial 5 is "+
               occurrence.result)
10         killMe
11     }
12 /* ... */
13 }
14
15 agent FactorialAgent {
16 /* ... */
17     on Factorial [ occurrence.number == upto ] {
18         info("Factorial of " + upto + " is " +
               occurrence.value)
19         emit(new ComputationDone=>[result=occurrence.
               value])
20         killMe
21     }
22 /* ... */
23     on Calculate {
24         this.upto = new Integer(occurrence.number);
```

```
25            info("Received Calculate for " + this.upto)
26            wake(new Factorial => [ number = 0 ; value = 1
                  ])
27        }
28    /* ... */
29    }
```

Example Code 3. Agent interaction in SARL

Now let's consider an example to illustrate the potential of extensibility of SARL to define specific interaction supports. For instance, to implement the organizational concepts of the CRIO metamodel [4], the developer can extend SARL by considering the organization as a particular type of Space Specification defining a new category of space where agents can interact only through the roles it plays. These roles can then be considered as a specialization of Behavior. The group instance of a given organization will therefore be considered as a Space, respecting the rules defined in its organizational specification, in which agent's identification is performed according to the different roles they play within the group.

## V. Recursive Agent and Hierarchical MAS with SARL, the Holonic Dimension

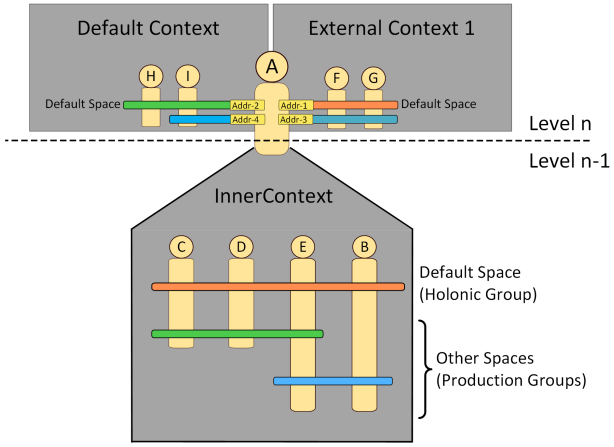


Fig. 4. A Holon or a recursive agent in SARL

In 1967, Arthur 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 self-similar holons as substructures [5]. Holonic Systems grew from the need to find comprehensive construct that could help explain social phenomena. Since then, it came to be used in a wide range of domains, including Philosophy [6], Manufacturing Systems [7], and Multi-Agents Systems [8].

Several works have studied this question and they have proposed a number of models inspired from their experience in different domains. In many cases we find the idea of *agents composed of other agents*. Each researcher gives a specific name to this type of agent, [9] discusses *individual* and *collective* agents; *meta-agents* are proposed by [10]; *Agentified Groups* are taken into account in the work of [11]; etc. All of these are only 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 [4] or O-MASE [12].

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 self-similar structures that compose each other via their Contexts. Each agent defines its own Context, called *Inner Context* and it is part of one or more *External Contexts*. For instance in Figure 4, agent A is taking part of two *External Contexts* (i.e. Default Context and External Context 1) and has its own *Inner Context* where agents B, C, D and E evolve.

*1) As part of a whole:* As mentioned previously, every agent in SARL evolves in a context. A context enables agents inside of it to interact via a *Default Space* and create new Spaces with specific interaction specifications. When an agent is spawned it will be created inside a Context that we call its *Default Context*. It is important to notice that the *Default Context* is not necessarily the same for every agent. It is defined at the agent's spawning by its parent. For instance, in Figure 4, if we assume agent B was spawned by agent A, A's *Inner Context* is B's *Default Context*.

At the top level, we consider an omnipresent agent (Root Agent). The VM will be in charge of spawning the first agents in the system as members of the Root Agent (see Section VI).

Agents can be part of more than one higher-level agents (called super-holon) at the same time. *ExternalContextAccess* BIC provides agents the mechanisms to `join`, `leave` and access external (super-holon's) contexts. The process and decision of how and when agents are accepted to become a member of the super-holon is left to the designer of the system. No specific procedure is imposed by SARL.

When an agent joins a new holon, SARL automatically register the agent in the *Default Space* of the holon's context. Therefore, all members of a super-holon are participants of the *Default Space*. This space becomes then a holonic structure where all agents' members of the same super-holon can interact, similar to the Holonic Group defined in [13].

It is important to notice, that Spaces belong to only one Context. This means that they are visible only to agents that have access to that context. For instance in Fig. 4 agent F cannot see, access or even know the Spaces in A's *Inner Context*. This is in fact one of holonic agent's definition fundamentals: A holon is seen from the outside as an atomic entity.

*2) As whole for its parts:* In fact, every agent defines a context where other agents can live. Therefore, every agent can be seen as a part of a larger holon (i.e. any SARL agent) and at the same time be composed by other holons that exist in its *Inner Context*.

Using Fig. 4 as reference, A is the super-holon of the agents B to E. An important question to answer is how A interacts or perceives it members. This can be achieved using the *InnerContextAccess* BIC. As its name implies, this capacity

gives access to agent's inner context and all the actions to modify it (e.g. create new spaces). Other BICs are constructed on top of it. The *Lifecycle* Capacity enables an agent to spawn new agents as peers or members depending on the provided *Default Context* for the new Agent.

Likewise, the *Behaviors* Capacity (discussed in Section III-A) uses the *Default Space* of the *InnerContext* as support for inter-behavior communication. This means that an event fired using the `wake` action will actually be emitted inside the *Default Space* of the *Inner Context*. Moreover, the *Behaviors* Capacity will listen to event in the *Default Space* of the *InnerContext* and show them as perceptions in the super-holon. Therefore, members' interactions in the *Default Space* of the *InnerContext* are perceived by their super-holon and any event fired by the super-holon's behaviors using `wake` can potentially raise reactions of its members.

## VI. Janus v2.0: a distributed runtime infrastructure of SARL

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 develop faster.

Nevertheless, we provide one of these infrastructures in the Janus Project. Janus is an open-source multi-agent platform fully implemented in Java 1.7. Janus version 2.0 was entirely rewritten to support SARL.

Janus implements all required infrastructure to execute a MAS programmed using SARL and fulfills its requirements such as fully distributed, parallel execution of agent's behaviors, automatic discovery of kernels, etc. Janus adopts best practices in current software development, such as Inversion of Control[8], and profits from new technologies like Distributed Data Structures[9]. The main purpose of Janus provides a VM for SARL MAS, and therefore provides implementations for all Built-in Capacities.

## VII. Related works

Authorship of APLs can be attributed to Shoham, who presented AOP as a new paradigm, a specialization of OOP promoting a societal view of computation where multiple agents interact one another [14]. In his vision, an agent is composed of mental states (beliefs, decisions, capabilities and obligations) and controlled by agents programs exploiting basic communication primitives. Since Shoham, a multitude of APLs based on different metamodels, formalisms and logics have been proposed. For obvious space reasons, it is impossible here to cite all of them, the reader can refer to [15] for a more comprehensive analysis. In this article, we limit our analysis only to the most notable languages excluding the agent platforms and frameworks like Jade, Janus, Jack, etc, since they do not provide first-class abstractions at the language level, but rather extend an existing object-oriented language usually Java.

One of the major trends that guided the creation of many APL is based on the concept of BDI agent (also true for many frameworks such as Jack). BDI [16] force agents to have a certain level of cognitive and reasoning capabilities, sometimes undesired in reactive MAS. From our point of view, BDI [16] remains a specific agent's architecture. Therefore, a generic APL must be independent of any agent's architecture and provides means to implement all of them. The integration within SARL of BDI-type or goal-based architectures will be undertaken in future works. According to our knowledge, SARL is the first general-purpose APL adopting this approach trying to remain architecture-independent and fully open. Within the BDI movement, four main APLs can be mentioned: ALOO [17] and SimpAL [18], 2APL [19] and 3APL [20], GOAL [21] and Jason-AgentSpeak [22]. All of them provide a set of tools to support software applications development. SimpAL[10] provides an Eclipse-based IDE including a compiler and a runtime support. The 2APL[11] programming language also comes with its corresponding execution platform and an Eclipse plug-in editor. SARL is generally in agreement with the definition of agent in simpAL as a state-full task- and event-driven entity. In SARL, agent's behaviors also communicates using an event-driven communication system and may schedule tasks using the *Schedule* BIC. However, the major difference is that SARL do not impose/set this approach, it is just one capacity among other available to agents to handle the notion of behavior's communication. Every developer can freely decide to develop a different approach and does not use this capacity. Unlike SARL, BDI-dependent APLs usually impose a fixed agent's control loop, like <sense>-<plan>-<act> in SimpAL, or the deliberation cycle of 3APL.

Within the architecture-independent APL, we may mention GAML agent-oriented language and its corresponding platform GAMA [23]. GAML is effectively architecture-independent but it currently focuses on multiagent-based simulation related issues and cannot now be considered as a general-purpose APL. Concerning the technology and the design, SARL and GAML shares the same development approach based on Xtext.

## VIII. Conclusion and future works

This article presents a new general purpose Agent-Oriented Programming Language named SARL. The main focus of SARL is to provide the minimum corpus of concepts required to define and develop MAS while remaining open and easily extensible. It also attempts to provide the community with a forum to study and compare various programming alternatives and metamodels. It is freely available for evaluation, extension and development under a permissive license[12].

The language offers an extensible and intuitive syntax for agent and MAS development. SARL recognizes the fact

[8] see https://code.google.com/p/google-guice/

[9] In-Memory Data Grid like Hazelcast: www.hazelcast.com

[10] http://simpal.sourceforge.net

[11] http://apapl.sourceforge.net

[12] http://www.apache.org/licenses/LICENSE-2.0

that the research community has provided a vast number agent architectures, interaction metamodels, communication and coordination mechanisms, etc; each with its own benefits. Therefore a general-purpose agent language should allow these models to be developed on top of it without imposing any specific architecture.

A first execution infrastructure is provided using the Janus platform. Janus allows agents to evolve in a fully distributed environment, with transparent network communication and concurrent execution of their behaviors.

SARL also has the associated tooling and IDE support for the language, making its development easier and with a shallow learning curve. Moreover, the language interoperability with Java let SARL profit from the advances provided by this community and simplifies legacy systems integration.

Future work will focus on mapping well known agent architectures (e.g. BDI [16]) and organizational models (e.g. MOISE [24] and CRIO [4]). We believe that the corpus of concepts provided in SARL will allow us to map these models into SARL concepts. Towards this end, work is ongoing to provide a first organizational extension for the language based on the CRIO metamodel [4].

## References


[1] S. Rodriguez, N. Gaud, V. Hilaire, S. Galland, and A. Koukam, "An analysis and design concept for self-organization in holonic multi-agent systems," in *the International Workshop on Engineering Self-Organizing Applications*. Springer-Verlag, May 2006, pp. 62–75.

[2] M. Cossentino, S. Galland, N. Gaud, V. Hilaire, and 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*, Venice, Italy, Oct. 2008.

[3] J. Peña, R. Levy, M. Hinchey, and A. Ruiz-Cortés, "Dealing with complexity in agent-oriented software engineering: The importance of interactions," in *Conquering Complexity*. Springer, 2012, pp. 191–214.

[4] M. Cossentino, N. Gaud, V. Hilaire, S. Galland, and 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*, vol. 2, no. 2, pp. 260–304, Mar. 2010.

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

[6] K. Wilber, *Sex, Ecology, Spirituality: The Spirit of Evolution*. Shambhala, 2000.

[7] E. van Leeuwen and D. Norrie, "Holons and holarchies," *Manufacturing Engineer*, vol. 76, no. 2, pp. 86–88, 1997.

[8] C. Gerber, J. Siekmann, and G. Vierke, "Holonic multi-agent systems," DFKI, Tech. Rep., 1999.

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

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

[11] J. Odell, M. Nodine, and R. Levy, "A metamodel for Agents, Roles, and Groups," in *Agent-Oriented Software Engineering V*, ser. LNCS, no. 3382. Springer, 2005, pp. 78–92.

[12] D. Case and S. DeLoach, "Applying an O-MaSE compliant process to develop a holonic multiagent system for the evaluation of intelligent power distribution systems," in *Engineering Multi-Agent Systems*, ser. LNCS, no. 8245. Springer, 2013, pp. 78–96.

[13] S. A. Rodriguez, "From analysis to design of holonic multi-agent systems: A framework, methodological guidelines and applications," Ph.D. dissertation, UTBM and UFC, 2005.

[14] Y. Shoham, "Agent-oriented programming," *Artificial Intelligence*, vol. 60, no. 1, pp. 51–92, Mar. 1993.

[15] R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, *Multi-Agent Programming: Languages, Tools and Applications*, 1st ed. Springer, 2009.

[16] A. S. Rao and M. P. Georgeff, "BDI agents: From theory to practice," in *In ICMAS*, 1995, pp. 312–319.

[17] A. Ricci and A. Santi, "Concurrent object-oriented programming with agent-oriented abstractions: The ALOO approach," in *AGERE*. ACM, 2013, pp. 127–138.

[18] ——, "Designing a general-purpose programming language based on agent-oriented abstractions: The simpAL project," in *the SPLASH'11 Workshops*. ACM, 2011, pp. 159–170.

[19] M. Dastani, "2APL: A practical agent programming language," *Autonomous Agents and Multi-Agent Systems*, vol. 16, no. 3, pp. 214–248, Jun. 2008.

[20] M. Dastani, M. B. Riemsdijk, F. Dignum, and J.-J. C. Meyer, "A programming language for cognitive agents goal directed 3APL," in *Programming Multi-Agent Systems*, ser. LNCS, 2004, vol. 3067, pp. 111–130.

[21] F. S. de Boer, K. V. Hindriks, W. van der Hoek, and J.-J. C. Meyer, "Agent programming with declarative goals," in *Intelligent Agents VII*, ser. LNCS, 2002, vol. 1986, pp. 117–131.

[22] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.

[23] A. Grignard, P. Taillandier, B. Gaudou, D. Vo, N. Huynh, and A. Drogoul, "GAMA 1.6: Advancing the art of complex agent-based modeling and simulation," in *PRIMA*, ser. LNCS, 2013, vol. 8291, pp. 117–131.

[24] M. Hannoun, O. Boissier, J. S. Sichman, and C. Sayettat, "MOISE: An organizational model for multi-agent systems," in *Advances in Artificial Intelligence*, ser. LNCS, no. 1952, 2000, pp. 156–165.