Semantic Transformation from SARL Agent-oriented Statements to Java Object-oriented Statements

Stéphane Galland[1] **and Sebastian Rodriguez**[2]

[1]CIAD, Univ. Bourgogne Franche-Comté, UTBM, F-90010 Belfort, France
stephane.galland@utbm.fr

[2]GITIA Lab, Universidad Tecnológica Nacional
San Miguel de Tucumán, CPA T4001JJD, Agentina
sebastian.rodriguez@gitia.org

ABSTRACT

As a general-purpose agent-oriented programming language, SARL aims at providing the fundamental abstractions that are usually considered as essential for implementing agent-based applications. Every programming language specifies an execution model. In the case of SARL, this execution model is defined based upon the object-oriented paradigm, i.e. a Java-based run-time environment. The goal of this paper is the explanation of the mapping between the agent paradigm and the object-oriented paradigm, and the definition of transformations from the SARL constructs to the object-oriented constructs. They enable the SARL developer understanding the SARL statements, and the mapping to executable entities.

Keywords: Agent-oriented Paradigm; Object-oriented Paradigm; Language Transformation; SARL.

2010 Mathematics Subject Classification: 68T42, 68T35, 97P40, 68M14.

Computing Classification System: Multi-agent systems; Domain specific languages; Distributed programming languages; Source code generation.

1 Introduction

During the last years, multiagent 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. Multiagent 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 (Cossentino, Galland, Gaud, Hilaire and Koukam, 2008), MaSE (DeLoach, 2004), or even Gaia (Wooldridge, Jennings and Kinny, 2000). In addition to these tools, several platforms have developed for agent oriented programming based on the main programming languages. There are several dozens, e.g. Jade

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

(Bellifemine, Caire and Greenwoord, 2007), NetLogo, (Wilensky, 1999) MATSim, (Horni, Nagel and Axhausen, 2016) or GAMA (Grignard, Taillandier, Gaudou, Vo, Huynh and Drogoul, 2013). These solutions are highly interesting because they frame and provide a methodology for the development of agent-based systems. On the other hand, these systems are very complex to implement, and the conventional programming languages 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. SARL[1] (Rodriguez, Gaud and Galland, 2014) is an agent-oriented programming language (APL) and a general purpose programming language (GPL) for implementing multiagent systems.

Every programming language specifies an execution model. Consequently, in order to execute a SARL program, a specific run-time environment must be defined. In this paper, the run-time environment is assumed to be written with an object-oriented language like Java. The goal of this paper is to *define the mapping from the SARL language constructs to the object oriented constructs.* In other words, a set of model transformations from the SARL metamodel to the Java metamodel is proposed. For example, this definition may be used for defining and running the Java equivalent programs from an SARL program on any Java-based agent framework, such as Janus (Gaud, Galland, Hilaire and Koukam, 2009), Jade (Bellifemine et al., 2007), or MATSim (Horni et al., 2016).

This paper is structured as follows. Section 2 explains the fundamentals of the SARL language. The formal definition of the transformations from SARL constructs to Java constructs is detailed in Section 3. Section 4 gives a complete example based on a simple interaction pattern. Section 5 provides a discussion on the related works. Finally, Section 6 concludes this paper and provides perspectives to this work.

2 SARL Agent-Programming Language

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.

Agent: According to Cossentino, Gaud, Hilaire, Galland and Koukam 2010, “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 as autonomy, reactivity, pro-activity and social capacities.” The various behaviors of an agent communicate using an event-driven approach by default.

Event: “An event is the specification of some occurrence in a space that may potentially trigger effects by a listener.” (Rodriguez et al., 2014)

Action: “An action is a specification of a transformation of a part of the designed system or its environment.” (Cossentino et al., 2010) This transformation guarantees resulting properties if the system before the transformation satisfies a set of constraints.

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

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

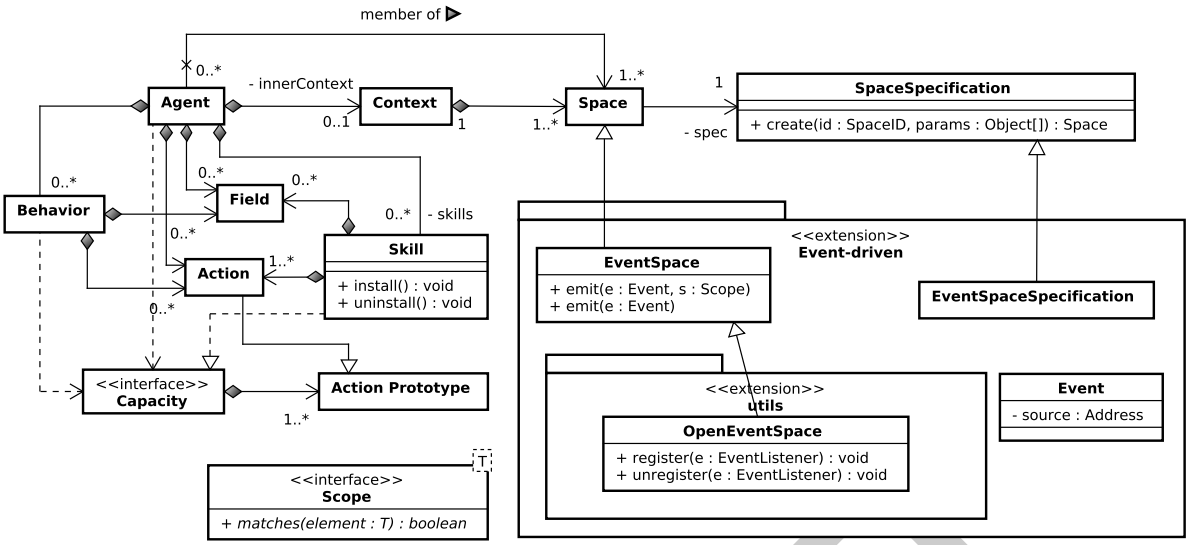

Figure 1: UML Class Diagram that presents the major concepts in the SARL metamodel.

Capacity: "A capacity is the specification of a collection of actions. This specification makes no assumptions about its implementation." (Cossentino et al., 2010) It could be used to specify what an agent can do, what a behavior requires for its execution.

Skill: "A skill is a possible implementation of a capacity fulfilling all the constraints of this specification." (Cossentino et al., 2010) An agent can dynamically evolve by learning/acquiring new capacities, but it can also dynamically change the skill associated to a given capacity (Rodriguez, Gaud, Hilaire, Galland and Koukam, 2006). Acquiring new skills 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.

3 Transformation Definitions

From a formal point of view, the transformation from the SARL language to the Java language may be specified by the notation proposed by Plotkin 2004 and Mosses 2004. This notation is selected because it enables the specification of context-aware transformations with a simple graphical formalism:

$$\frac{C}{\langle A, \sigma \rangle \rightarrow \langle B, \sigma' \rangle}$$

The transformation context is represented by σ and σ', respectively before and after the application of the transformation. The previous expression means: when A is a candidate for transformation within the context σ, and the conditions in term C are true, then A is transformed to B, and the context becomes σ'. It is assumed that two different rules cannot have their A evaluated to true on the same σ. All the transformations described in this section are implemented in the SARL compiler.

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

3.1 Type Declaration Transformation

The four constructs for type declaration are for events, capacities, skills and agents.

3.1.1 Event

Definition 3.1. An event e is the specification of some occurrence in a space that may potentially trigger effects by a listener. An event e is defined inside the set $\mathbb{E}$ of all the events by its identifier id_e, its set F_e of fields (or attributes) and its set B_e of construction functions. The definition of the event e may extend a previously defined event $s_e \in \mathbb{E}$ with $s_e \neq e$[2].

$$e = \langle id_e, s_e, F_e, B_e \rangle$$

Equations 3.1 to 3.4 detail the transformations to apply on an SARL event construct for obtaining the Java construct. The modeling of an event in Java is the first major question arising. In Java, objects contain data, in the form of fields; and code, in the form of methods (Ghezzi, Jazayeri and Mandrioli, 2002). Equivalent construct in Java to an SARL event is an object that contains the same fields. This modeling approach is widely assumed in computer programming, such as in the event-based simulation (Fritzson and Bunus, 2002), component-oriented programming (Szyperski, Bosch and Weck, 1999), object-oriented database (Gehani, Jagadish and Shmueli, 1992), or event-based notification in software programs (Matheny, White, Anderson and Schaeffer, 2002). Equation 3.1 describes the transformation of an event declaration without field definition. The Java equivalent type extends the `Event` type, which is defined in the SARL libraries. This type has two roles: (i) providing the functions that are common to all the events. For example, the `getSource` method is defined for replying to the identity of the agent that fired the event; and (ii) defining a super root type for all the SARL events at the Java layer. In Equation 3.1, the state is expanded with the definition of the event e[3].

$$\frac{e \notin \mathbb{E} \qquad \sigma' = \sigma[\mathbb{E}/\mathbb{E}; e]}{\langle \textbf{event } id_e, \sigma \rangle \rightarrow \langle \textbf{class } id_e \textbf{ extends } \texttt{Event}\{\}, \sigma' \rangle} \tag{3.1}$$

Equation 3.2 refines the previous equation by enabling the explicit declaration of the type that the event extends. Function $superType(x) : \mathbb{U} \mapsto \mathcal{P}()U$ retrieves all the super-types of its argument x.

$$\frac{\begin{array}{cc} e \notin \mathbb{E} & s_e \in \{id_x | x \in \mathbb{E}\} \\ id_e \notin superTypes(s_e) & \sigma' = \sigma[\mathbb{E}/\mathbb{E}; e] \end{array}}{\begin{array}{c} \langle \textbf{event } id_e \textbf{ extends } s_e, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_e \textbf{ extends } s_e \ \{\}, \sigma' \rangle \end{array}} \tag{3.2}$$

Equations 3.3 and 3.4 extend respectively Equations 3.1 and 3.2 by enabling the definition of B for the event e. B is composed by the set F_e of the fields of e, and the set B_e of the construction functions of e. In the rest of this paper, they are known as the members of e.

[2] Inheritance definition between the SARL language types is outside the scope of this paper.

[3] The notation $a[b/c]$ means: a copy of a in which b is replaced by c.

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

$$\frac{e \notin \mathbb{E} \quad B = F_e \cup B_e \rightarrow B' \quad \sigma' = \sigma[\mathbb{E}/\mathbb{E}; e]}{\langle \textbf{event } id_e \{ B \}, \sigma \rangle \rightarrow \langle \textbf{class } id_e \textbf{ extends } \texttt{Event}\{ B' \}, \sigma' \rangle} \tag{3.3}$$

$$\frac{\begin{array}{c} e \notin \mathbb{E} \quad s_e \in \{id_x | x \in \mathbb{E}\} \quad id_e \notin superTypes(s_e) \\ B = F_e \cup B_e \rightarrow B' \quad \sigma' = \sigma[\mathbb{E}/\mathbb{E}; e] \end{array}}{\langle \textbf{event } id_e \textbf{ extends } s_e \{ B \}, \sigma \rangle \rightarrow \langle \textbf{class } id_e \textbf{ extends } s_e \{ B' \}, \sigma' \rangle} \tag{3.4}$$

3.1.2 Capacity

Definition 3.2. A capacity c is the specification of a collection of actions. A capacity c is defined inside the set $\mathbb{C}$ of all the capacities by its identifier id_c, and its set M_c of body-less functions. A capacity c could extend several previously defined capacities $s_c \in S_c \subseteq \mathbb{C}$ with $s_c \neq e$.

$$c = \langle id_c, S_c, M_c \rangle$$

Equivalent construct in Java to an SARL capacity is an interface. Indeed, a capacity defines a collection of functions' prototypes, as the interface concept. Equation 3.5 describes the transformation of a capacity declaration when no extended capacity type is specified. The `Capacity` type is implicitly assumed as the root type for all the capacities. Equation 3.6 defines the transformation when an extended capacity type is explicitly defined.

$$\frac{c \notin \mathbb{C} \quad M_c \rightarrow M_c' \quad \sigma' = \sigma[\mathbb{C}/\mathbb{C}; c]}{\langle \textbf{capacity } id_c \{ M_c \}, \sigma \rangle \rightarrow \langle \textbf{interface } id_c \textbf{ extends } \texttt{Capacity}\{ M_c' \}, \sigma' \rangle} \tag{3.5}$$

$$\frac{\begin{array}{c} c \notin \mathbb{C} \quad S_c \subseteq \{id_x | x \in \mathbb{C}\} \quad id_c \notin superTypes(S_c) \\ M_c \rightarrow M_c' \quad \sigma' = \sigma[\mathbb{C}/\mathbb{C}; c] \end{array}}{\langle \textbf{capacity } id_c \textbf{ extends } S_c \{ M_c \}, \sigma \rangle \rightarrow \langle \textbf{interface } id_c \textbf{ extends } S_c \{ M_c' \}, \sigma' \rangle} \tag{3.6}$$

3.1.3 Skill

Definition 3.3. A skill s is a possible implementation of a capacity c_s. A skill s is defined inside the set $\mathbb{S}$ of all the skills by its identifier id_s, its set F_s of fields, its set M_c of functions, its set B_s of construction functions, its set C_s of implemented capacities, and its extended type s_s.

$$s = \langle id_s, s_s, C_s, F_s, M_s, B_s \rangle$$

Because the capacity concept is mapped to a Java interface, and a skill is implementing such a capacity, the skill concept is mapped to a ava class. Equations 3.7 and 3.8 describe the transformations of a skill declaration, without and with the specification of an extended type, respectively. As for the event and the capacity concepts, a root Java type is defined in the SARL library; and it is implicitly used as the extended type when this latest is not specified.

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

$$\frac{\begin{array}{c} s \notin \mathbb{S} \qquad I_s \subseteq \{id_x | x \in \mathbb{C}\} \\ B = F_s \cup B_s \cup M_s \cup H_s \rightarrow B' \qquad \sigma' = \sigma[\mathbb{S}/\mathbb{S}; s] \end{array}}{\begin{array}{c} \langle \textbf{skill } id_s \textbf{ implements } I_s \{ B \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_s \textbf{ extends } \texttt{Skill} \textbf{ implements } I_s \{ B' \}, \sigma' \rangle \end{array}} \tag{3.7}$$

$$\frac{\begin{array}{c} s \notin \mathbb{S} \quad I_s \subseteq \{id_x | x \in \mathbb{C}\} \quad id_s \notin superTypes(S_s) \\ B = F_s \cup B_s \cup M_s \cup H_s \rightarrow B' \qquad \sigma' = \sigma[\mathbb{S}/\mathbb{S}; s] \end{array}}{\begin{array}{c} \langle \textbf{skill } id_s \textbf{ extends } S_s \textbf{ implements } I_s \{ B \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_s \textbf{ extends } S_s \textbf{ implements } I_s \{ B' \}, \sigma' \rangle \end{array}} \tag{3.8}$$

3.1.4 Agent

Definition 3.4. An agent a is an autonomous entity having a set of skills to realize the capacities it exhibits. An agent a is defined inside the set $\mathbb{A}$ of all the agents by its identifier id_a, its set F_a of fields, its set M_a of functions, its set B_a of construction functions, its set H_a of event handlers, and its extended type s_b.

$$a = \langle id_a, s_a, F_a, M_a, B_a, H_a \rangle$$

Additionally, an agent a may have behaviors DB_a and skills S_a that could be dynamically being registered by the agent. There two sets are supported by the root type `Agent` in the SARL library. The DB_a and S_a sets are not included within the definition of the agent a above; because they are dynamic properties of the agent; and no specific statements are provided in the SARL language: the functions inherited from `Agent` enable to manage them. Equations 3.9 and 3.10 provide the transformations for the **agent** construct.

$$\frac{\begin{array}{c} a \notin \mathbb{A} \quad B = F_a \cup B_a \cup M_a \cup H_a \rightarrow B' \\ \sigma' = \sigma[\mathbb{A}/\mathbb{A}; a] \end{array}}{\begin{array}{c} \langle \textbf{agent } id_a \{ B \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_a \textbf{ extends } \texttt{Agent} \{ B' \}, \sigma' \rangle \end{array}} \tag{3.9}$$

$$\frac{\begin{array}{c} a \notin \mathbb{A} \quad s_a \in \{id_x | x \in \mathbb{A}\} \quad id_a \notin superTypes(s_a) \\ B = F_a \cup B_a \cup M_a \cup H_a \rightarrow B' \qquad \sigma' = \sigma[\mathbb{A}/\mathbb{A}; a] \end{array}}{\begin{array}{c} \langle \textbf{agent } id_a \textbf{ extends } s_a \{ B \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_a \textbf{ extends } s_a \{ B' \}, \sigma' \rangle \end{array}} \tag{3.10}$$

4 Example: Ping-Pong Application

In order to illustrate the transformation from the SARL language to the Java language, a simple application example is used. Two types of agents are considered: `Initiator` and `Answerer`. The `Initiator` agent sends a `Ping` event to the `Answerer` agent. When this latest agent has received the event, it sends a `Pong` event back to the `Initiator` agent. When this latest agent has received a `Pong` event, it writes a message to the computer console.

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

4.1 Event Definition

The two events `Ping` and `Pong` are defined in Listing 1. In order to trace the event exchanges, each event is identified by an integer number. This identifier is used for building the message to be logged on the console.

```
1  event Ping {
2    var identifier : int
3    new (id : int) {
4      identifier = id
5    }
6  }
7  event Pong {
8    var identifier : int
9    new (originalEvent : Ping) {
10     identifier = id.identifier
11   }
12 }
```

Listing 1: Definition of the `Ping` and `Pong` events.

Regarding the `Ping` event, its identifier is defined as an integer field that is publicly accessible — the default accessibility of the events' members – at the line 2 of Listing 1. This field is initialized in the constructor member of the event (Line 4). Regarding the `Pong` event, its identifier has the same value as the identifier of the corresponding `Ping` event. At Line 10, the `Ping` event identifier is copied into the `Pong` event.
The application of the above equations on the `Ping` event produces the Java program in Listing 2. The transformation of a field and a construction function are both illustrated by this example. The Java code for the `Pong` event is similar, so that it is not detailed in this paper.

```
1  public class Ping extends Event {
2    public int identifier;
3    public Ping(final int id) {
4      this.identifier = id;
5    }
6  }
```

Listing 2: Java program for the the `Ping` event.

4.2 Logging Capacity and Skill Definitions

The `Initiator` agent should write a message on the standard computer console. Writing a message is a capability of the agent. It is defined and declared with an SARL capacity at Line 1 of Listing 3. This capacity enables the access to the function at Line 2.

```
1  capacity Log {
2    def showMessage(message : String)
3  }
4  skill ConsoleLog implements Log {
5    def showMessage(message : String) {
6      System::out.println(message)
7    }
```


Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

```
8 }
```

Listing 3: Definition of the `Log` capacity and `ConsoleLog` skill.

Each capacity should have a least one associated implementation, i.e. a skill. The `showMessage` method is defined in order to write on the computer console (Line 5). The `System` type and the `println` method are part of the standard Java library, which is usable from any SARL program (Rodriguez et al., 2014).
Listing 4 shows the Java program that is equivalent to the SARL program in Listing 3.

```
1 public interface Log extends Capacity {
2   void showMessage(String message);
3 }
4 public class ConsoleLog extends Skill implements Log {
5   public void showMessage(String message) {
6     System.out.println(message);
7   }
8 }
```

Listing 4: Java program for the the `Log` capacity and the `ConsoleLog` skill.

4.3 Initiator Agent Definition

The behavior of the `Initiator` agent is divided into three sub-behaviors. Firstly, the agent must learn the skill that is defined in the previous section. According to the SARL specification, each agent has a least access to the `setSkill` method that registers a skill to the calling agent. This function is invoked at Line 4 for providing the `Initiator` agent with an `ConsoleLog`. As soon as the `setSkill` is called, all the methods that are defined inside could be explicitly called from the agent's code. The **uses** statement at Line 2 makes the methods from the specified capacities accessible from everywhere in the agent's code. Lines 6 and 9 illustrate two calls to the capacities' methods: `emit` from the `DefaultContextInteractions` capacity, and `showMessage` from the `Log` capacity.

```
1  agent Initiator {
2    uses DefaultContextInteractions, Log
3    on Initialize {
4      setSkill(new ConsoleLog)
5      var pingEvent = new Ping(0)
6      emit(pingEvent)
7    }
8    on Pong {
9        showMessage("I have received the answer for event "
10           + occurrence.identifier)
11   }
12 }
```

Listing 5: Definition of the `Initiator` agent type.

The second part of the `Initiator` agent's behavior contains the emitting of the `Ping` event. An instance of this event is created, and fired into the default communication space at Line 6.

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

The third part of the agent's behavior concerns the reception of the `Pong` event. This part is supported by the event handler, which is specified after the **on** keyword. This handler is automatically executed when the agent receives an occurrence of the `Pong` event. The code of the handler writes a message with the `Log` capacity. The **occurrence** keyword represents the instance of the event for which the event handler is executed.
Two event handlers are defined within the SARL program. Each of them has an associated Java function for supporting the event handler instructions (Lines 2 and 8 in Listing 6). For each event that is handled by the agent, a function for the guard evaluation is generated within the Java program (Lines 13 and 17). Because there is no explicit guard, the event handlers are always added to the collection of the runnable event handlers. The **uses** keyword in the SARL program causes the addition of all the declared capacities' functions to the scope of the agent's code. It means that each reference to a function of a "used" capacity of the SARL program is transformed to a direct invocation of the function on the skill instance into the Java program, as illustrated at Line 6. The `setSkill` and `getSkill` functions are provided by inheritance by the `Agent` supertype.

```
1  public class Initiator extends Agent {
2    private void $on$io_sarl_core_Initialize$0(
3      final Initialize occurrence) {
4      setSkill(new ConsoleLog());
5      Ping pingEvent = new Ping(0);
6      getSkill(DefaultContextInteractions.class).emit(pingEvent);
7    }
8    private void $on$Pong$0(final Pong occurrence) {
9      getSkill(Log.class).showMessage(
10       "I have received the answer for event "
11       + occurrence.identifier);
12   }
13   public void $eval$io_sarl_core_Initialize(
14       final Initialize occurrence, List<Runnable> handlers) {
15     handlers.add(() ->$on$io_sarl_core_Initialize$0(occurrence));
16   }
17   public void $eval$Pong(final Initialize occurrence,
18       List<Runnable> handlers) {
19     handlers.add(() ->$on$Pong$0(occurrence));
20   }
21 }
```

Listing 6: Java program for the the `Initiator` agent.

The `Answerer` agent is defined in Listing 7 in order to send a `Pong` event with the same identifier of a received `Ping` event. The emitting of the `Pong` event at Line 5 is scoped in order to restrict the receiver of the events to the same agent that has sent the `Ping` event. The scoping expression is expressed with a lambda expression, which takes the address as the receiver as argument (the **it** keyword).

```
1  agent Answerer {
2    uses DefaultContextInteractions
3    on Ping {
4      var pongEvent = new Pong(occurrence)
```


Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

```
5       emit(pongEvent) [it == occurrence.source]
6     }
7   }
```

Listing 7: Definition of the `Answerer` agent type.

Listing 8 provides the Java program that is equivalent to the SARL program in Listing 7. The Java code provides a concrete example of the support of the closures, a.k.a. as lambda expressions; and how they are translated from SARL to Java.

```
1  public class Answerer extends Agent {
2    private void $on$Ping$0(final Ping occurrence) {
3      Pong pongEvent = new Pong(occurrence);
4      getSkill(DefaultContextInteractions.class).emit(pongEvent,
5        (it) -> it.equals(occurrence.getSource()));
6    }
7  }
```

Listing 8: Java program for the the `Answerer` agent.

The source code of this example of transformation, and other examples could be download for the SARL server.

4.4 Execution of the Ping-Pong Application

Generating the target code from the SARL code is not enough to obtain a fully runnable application. According to the tool-chain of SARL, the ping-pong application is run by the SARL Runtime Environment (SRE). The SRE has the same role for SARL as the Java virtual machine for Java: it runs the application from compiled source code. Figure 2 shows a UML sequence diagram of a run of the application. The SRE creates the `Initiator` and the `Answerer` agents, respectively by the interactions 1 and 4. Interaction 1 results in the initialization of the `Initiator` with the asynchronous invocation of the `Initialize` event handler (Interactions 2 and 3). No event handler is invoked into the `Answerer` agent because no specific code was generated in Listing 8.

According to Listing 6, the `Initiator` agent creates an instance of `Ping` event (Interaction 3.1) in the `Initialize` event handler. Then, it sends this event asynchronously to the other agent through the SRE (Interaction 3.3) after retrieving the agent's capacity that supports the agent communication (Interaction 3.2).

When the SRE receives a query to route an event, it finds the receiving agent(s) and invokes asynchronously the event handler(s). In the case of the ping-pong application, Interactions 3.3.1 and 3.3.2 correspond to the routing of a `Ping` event to the `Answerer` agent.

When the `Answerer` agent receives a `Ping` event (Interaction 3.3.2), it creates an instance of the `Pong` event (Interaction 3.3.2.1). Then, it sends this event asynchronously to the `Initiator` agent (Interaction 3.3.2.3), after retrieving the agent's capacity that supports the agent communication (Interaction 3.3.2.2). The SRE delivers asynchronously the event to the `Initiator` agent (Interaction 3.3.2.3.2).

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

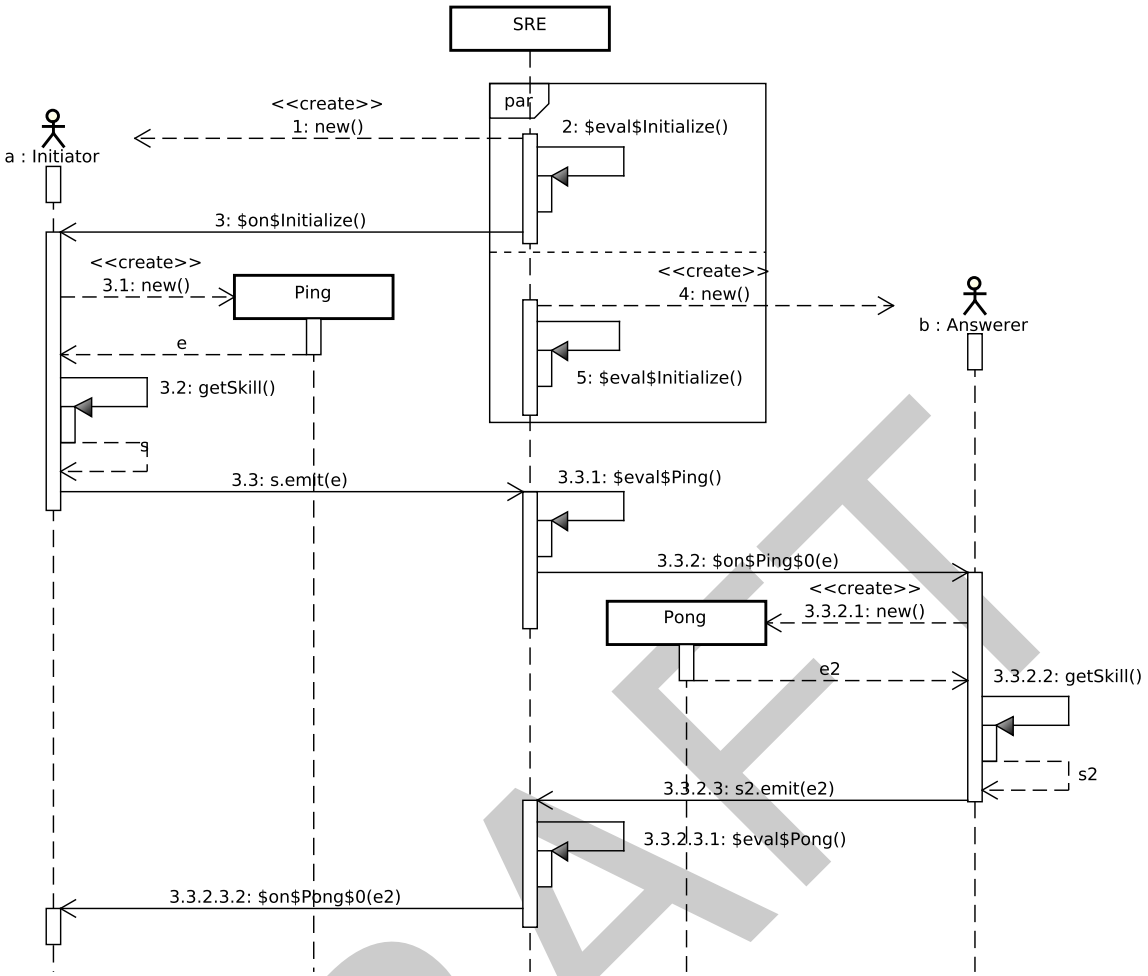

Figure 2: UML sequence diagram for a run of the Ping-Pong application.

5 Related Works

Since Shoham 1993, a multitude of APL based on different metamodels, formalisms and logics have been proposed. For obvious space reasons, it is impossible to cite all of them; the reader can refer to (Bordini, Dastani, Dix and Seghrouchni, 2009) for a more comprehensive analysis. One of the major trends that guided the creation of many APL is the concept of BDI agent. BDI (Rao and Georgeff, 1995) forces agents to have a certain level of cognitive and reasoning capabilities, sometimes undesired in reactive MAS. From our point of view, BDI remains a specific agent's architecture. A generic APL must be independent of any agent's architecture and provide means to implement all of them. The integration within SARL of BDI 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. Four main APL that are including BDI can be mentioned: SimpA (Ricci, Viroli and Piancastelli, 2011) or SimpAL (Ricci and Santi, 2011), 2APL (Dastani, 2008) or 3APL (Dastani, Riemsdijk, Dignum and Meyer, 2004), GOAL (de Boer, Hindriks, van der Hoek and Meyer, 2002) and Jason/AgentSpeak (Bordini, Hübner and Wooldridge, 2007). They also provide tools to support soft-

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

ware application's development. SimpAL provides an Eclipse-based IDE, including a compiler and a run-time support. The 2APL programming language also comes with its corresponding execution platform and an Eclipse plug-in editor. SARL is generally in agreement over the definition of agent in SimpAL as a state-full task-driven and event-driven entity. In SARL, agents also communicate using an event-driven approach, and schedule tasks using the *Schedule* BIC. However, SARL does not impose this approach in opposite to the previously mentioned languages. Even-driven communication is one capacity among others to handle the notion of communication. Every developer can freely decide to develop a different approach and does not use this capacity. Unlike SARL, a BDI-dependent APL usually imposes a fixed agent's control loop, like <sense>-<plan>-<act> in SimpAL, or the deliberation cycle of 3APL.
Glake, Weyl, Dohmen, Hüning and Clemen 2017 proposed an agent-based domain specific language (DSL) upon the MARS framework. Its set of features and those from SARL have an not-empty intersection, such as the specification of reactive behaviors. Nevertheless, MARS has the capability to define probes on agents' attributes, SARL not. On the other side, SARL enables to implement hierarchical systems, and managed distribution other a computer network, MARS not.
Within the architecture-independent APL, we may mention GAML agent-oriented language and its platform GAMA (Grignard et al., 2013). GAML focuses on multiagent-based simulation, and cannot be considered as a general-purpose APL. SARL and GAML are both implemented with the Xtext framework.

6 Conclusion and Perspectives

The SARL 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, and many implement at least part of that model to a run-time system. In the case of SARL programs, the SRE provides the tools and the features that are mandatory for running such a program according to the SARL specifications. This paper regards the SRE to be a Java-based application. The transformations of the SARL constructs are described to be equivalent to Java's constructs. These transformation rules may be adapted in order to target another language, object-oriented or not.
Future work will focus on mapping well-known agent architectures e.g. BDI (Rao and Georgeff, 1995), organizational models e.g. MOISE (Hannoun, Boissier, Sichman and Sayettat, 2000) or CRIO (Cossentino et al., 2010). It can be believed that the corpus of concepts provided in SARL can be allowed 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. Additionally, a SARL program may be transformed into another language other than Java. The transformation to the Python, C# and C languages is under study and implementation. They are specifically used for deploying an agent program on specific platforms, such as machine learning environment or embedded system. Finally, other agent frameworks such as Jade (Bellifemine et al., 2007) or MATSIM (Illenberger, Flötteröd and Nagel, 2007) will be

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

considered as possible targets of the SARL program transformation.

References

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

Bordini, R. H., Dastani, M., Dix, J. and Seghrouchni, A. E. F. 2009. *Multi-Agent Programming: Languages, Tools and Applications*, 1st edn, Springer Publishing Company, Incorporated.

Bordini, R. H., Hübner, J. F. and Wooldridge, M. 2007. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*, John Wiley & Sons.

Cossentino, M., Galland, S., Gaud, N., Hilaire, V. and Koukam, A. 2008. How to control emergence of behaviours in a holarchy, *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-Organizaing Systems (SASO 2008)*, IEEE Computer Society, Venice, Italy.

Cossentino, M., Gaud, N., Hilaire, V., Galland, S. and Koukam, A. 2010. 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): 260–304.

Dastani, M. 2008. 2APL: A practical agent programming language, *Autonomous Agents and Multi-Agent Systems* **16**(3): 214–248.

Dastani, M., Riemsdijk, M. B., Dignum, F. and Meyer, J.-J. C. 2004. A programming language for cognitive agents goal directed 3APL, *in* M. Dastani, J. Dix and A. El Fallah-Seghrouchni (eds), *Programming Multi-Agent Systems*, Vol. 3067 of *LNCS*, Springer Berlin Heidelberg, pp. 111–130.

de Boer, F. S., Hindriks, K. V., van der Hoek, W. and Meyer, J.-J. C. 2002. Agent programming with declarative goals, *CoRR* **cs.AI/0207008**.

DeLoach, S. A. 2004. *The MaSE Methodology*, Springer US, Boston, MA, pp. 107–125.

Fritzson, P. and Bunus, P. 2002. Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation, *Proceedings 35th Annual Simulation Symposium. SS 2002*, pp. 365–380.

Gaud, N., Galland, S., Hilaire, V. and Koukam, A. 2009. An organizational platform for holonic and multiagent systems, *in* K. Hindriks, A. Pokahr and S. Sardina (eds), *6th International Workshop ProMAS 2008, LNCS 5442*, Springer Berlin Heidelberg, Estoril, Portugal, pp. 104–119.

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

Gehani, N. H., Jagadish, H. V. and Shmueli, O. 1992. Event specification in an active object-oriented database, *SIGMOD Rec.* **21**(2): 81–90.
URL: *http://doi.acm.org/10.1145/141484.130300*

Ghezzi, C., Jazayeri, M. and Mandrioli, D. 2002. *Fundamentals of Software Engineering*, 2nd edn, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Glake, D., Weyl, J., Dohmen, C., Hüning, C. and Clemen, T. 2017. Modeling through model transformation with MARS 2.0, *International Springer Simulation Conference*. DOI: 10.22360/springsim.2017.ads.005.

Grignard, A., Taillandier, P., Gaudou, B., Vo, D., Huynh, N. and Drogoul, A. 2013. GAMA 1.6: Advancing the art of complex agent-based modeling and simulation, *in* G. Boella, E. Elkind, B. Savarimuthu, F. Dignum and M. Purvis (eds), *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, Vol. 8291 of *LNCS*, Springer Berlin Heidelberg, pp. 117–131.

Hannoun, M., Boissier, O., Sichman, J. S. and Sayettat, C. 2000. MOISE: An organizational model for multi-agent systems, *in* M. C. Monard and J. S. Sichman (eds), *Advances in Artificial Intelligence*, number 1952 in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 156–165.

Horni, A., Nagel, K. and Axhausen, K. 2016. *The Multi-Agent Transport Simulation MATSim*, Ubiquity Press Limited.

Illenberger, J., Flötteröd, G. and Nagel, K. 2007. Enhancing MATSim with capabilities of within-day re-planning, *IEEE Intelligent Transportation Systems Conference*, Seattle, WA, USA, pp. 94–99.

Matheny, J., White, C., Anderson, D. and Schaeffer, A. 2002. Object-oriented event notification system with listener registration of both interests and methods. US Patent 6,424,354.
URL: *https://www.google.com/patents/US6424354*

Mosses, P. 2004. Exploiting labels in structural operational semantics, *Fundam. Inform.* **60**(1–4): 17–31.

Plotkin, G. 2004. A structural approach to operational semantics, *J. Log. Algebr. Program* **60–61**: 17–139.

Rao, A. S. and Georgeff, M. P. 1995. Bdi agents: From theory to practice, *In proceedings of the first international conference on multi-agent systems (ICMAS-95*, pp. 312–319.

Ricci, A. and Santi, A. 2011. Designing a general-purpose programming language based on agent-oriented abstractions: The simpal project, *Proceedings of the SPLASH'11 Workshops*, ACM, New York, NY, USA, pp. 159–170.

Ricci, A., Viroli, M. and Piancastelli, G. 2011. simpa: An agent-oriented approach for programming concurrent applications on top of java, *Sci. Comput. Program.* **76**(1): 37–62.

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.

Rodriguez, S., Gaud, N. and Galland, S. 2014. SARL: A general-purpose agent-oriented programming language, *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on*, Vol. 3, pp. 103–110.

Rodriguez, S., Gaud, N., Hilaire, V., Galland, S. and Koukam, A. 2006. An analysis and design concept for self-organization in holonic multi-agent systems, *the International Workshop on Engineering Self-Organizing Applications (ESOA'06)*, Springer-Verlag, pp. 62–75.

Shoham, Y. 1993. Agent-oriented programming, *Artificial Intelligence* **60**(1): 51–92.

Szyperski, C., Bosch, J. and Weck, W. 1999. *Component-Oriented Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 184–192.

Wilensky, U. 1999. Netlogo, *Technical report*, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Wooldridge, M., Jennings, N. R. and Kinny, D. 2000. The gaia methodology for agent-oriented analysis and design, *Autonomous Agents and Multi-Agent Systems* **3**(3): 285–312.

Accepted by International Journal of Artificial Intelligence, vol. 17(2), pp. 139-153.