Transformations from the SARL agent-oriented language to the Java object-oriented language

Stéphane GALLAND
LE2I
Univ. Bourgogne Franche-Comté, UTBM
F-90010 Belfort, France
stephane.galland@utbm.fr

Sebastian RODRIGUEZ
GITIA
Universidad Tecnológica Nacional
San Miguel de Tucumán, CPA T4001JJD, Agentina
sebastian.rodriguez@gitia.org

***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. In the case of SARL, this execution model is defined based upon the object-oriented paradigm, e.g. a run-time environment written with Java. Accordingly, and by default, the SARL programs are transformed into their equivalent object-oriented programs. 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 standard object-oriented constructs. They enable the SARL developer understanding the SARL statements, and the mapping to executable entities. The transformations in this paper could also serve as the basis for creating a compiler extension for targeting any object-oriented programming language.**

***Index Terms*—Agent-oriented Paradigm; Object-oriented Paradigm; Language Transformation; SARL**

I. 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, MaSE, or even Gaia.In addition to these tools, the main programming languages have developed platforms for agent oriented programming. There are several dozens, e.g. Jade,NetLogo, MATSim, or GAMA.

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 (Rodriguez et al., 2014) is a new general-purpose agent-oriented programming language (APL) for setting up adaptive and modular principles for developing multiagent systems.

Every programming language specifies an execution model, and many implement at least part of that model into a run-time system. For an SARL program being executed, a specific run-time environment must be defined. In this paper, we assume that the run-time environment is written with an object-oriented language, e.g. Java. The goal of this paper is to *define the mapping from the SARL language constructs to the object oriented constructs.* In other words, the model transformation 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, Jade, or MATSim.

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

II. SARL Agent-Programming Language

SARL[1] is a general-purpose agent-oriented programming language (Rodriguez et al., 2014). This language aims at providing the fundamental abstractions for dealing with 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.

Agent: 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

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

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.

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.

Capacity: 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.

Skill: 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. 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.

III. Transformation Definitions

In this section, the transformations that should be applied to the SARL constructs in order to obtain the Java equivalent constructs are detailed. From a formal point of view, the transformation from the source language, SARL, to the target 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'. All the transformations described in is section are implemented in the SARL compiler[2].

A. Type Declaration Transformation

In SARL, four type declaration constructs are available. The corresponding types are the events, the capacities, the skills, and the agents.

1) Event:

Definition 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 $s_e \neq e$[3].

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

Equations 1 to 4 detail the transformations to apply on an SARL event construct for obtaining the Java language equivalent 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 procedures, often known as methods (Ghezzi et al., 2002). Equivalent construct in Java to an SARL event is an object that contains the same fields as in the SARL event. This modeling approach is widely assumed in computer programming, e.g. event-based simulation (Fritzson and Bunus, 2002), component-oriented programming Szyperski et al. (1999), object-oriented database (Gehani et al., 1992), or event-based notification in software programs (Matheny et al., 2002). Equation 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[4]. This type has two roles: (i) providing the methods that are shared by all the events. For example, the `getSource` method is defined for replying to the identity of the emitter of the event; and (ii) defining a super root type for all the SARL events at the Java layer. In Equation 1, the state is expanded with the definition of the event e[5].

$$\frac{e \notin \mathbb{E} \qquad \sigma' = \sigma[\mathbb{E}/\mathbb{E}; e]}{\langle \textbf{event } id_e, \sigma\rangle \rightarrow \langle \textbf{class } id_e \textbf{ extends } \texttt{Event}\{\}, \sigma'\rangle} \tag{1}$$

Equation 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.

$$\frac{\begin{array}{cc} e \notin \mathbb{E} & s_e \in \{id_x | x \in \mathbb{E}\} \\ id_e \notin superTypes(s_e) & \sigma' = \sigma[\mathbb{E}/\mathbb{E}; e]\end{array}}{\begin{array}{c}\langle \textbf{event } id_e \textbf{ extends } s_e, \sigma\rangle \rightarrow \\ \langle \textbf{class } id_e \textbf{ extends } s_e\ \{\}, \sigma'\rangle\end{array}} \tag{2}$$

Equations 3 and 4 extend respectively Equations 1 and 4 by enabling the declaration of the content B for the event e. This content 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.

$$\frac{e \notin \mathbb{E} \qquad B = F_e \cup B_e \rightarrow B' \qquad \sigma' = \sigma[\mathbb{E}/\mathbb{E}; e]}{\begin{array}{c}\langle \textbf{event } id_e\ \{\ B\ \}, \sigma\rangle \rightarrow \\ \langle \textbf{class } id_e \textbf{ extends } \texttt{Event}\{\ B'\ \}, \sigma'\rangle\end{array}} \tag{3}$$

[2]SARL Model Inferrer: https://github.com/sarl/sarl/main/coreplugins/io.sarl.lang/src/io/sarl/lang/jvmmodel/SARLJvmModelInferrer.java

[3]Inheritance definition between the SARL language constructs is outside the scope of this paper.

[4]`Event` in the SARL Library: http://www.sarl.io/docs/api/io/sarl/lang/core/Event.html.

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

$$\frac{\begin{array}{c} e \notin \mathbb{E} \quad s_e \in \{id_x | x \in \mathbb{E}\} \quad id_e \notin superTypes(s_e) \\ B = F_e \cup B_e \rightarrow B' \quad \sigma' = \sigma[\mathbb{E}/\mathbb{E}; e] \end{array}}{\begin{array}{c} \langle \textbf{event } id_e \textbf{ extends } s_e \ \{ \ B \ \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_e \textbf{ extends } s_e \ \{ \ B' \ \}, \sigma' \rangle \end{array}} \tag{4}$$

2) Capacity:

Definition 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 5 describes the transformation of a capacity declaration when no extended capacity type is specified. In this case, the `Capacity` type[6] is implicitly assumed as the root type for all the capacities. Equation 6 defines the transformation when an extended capacity type is explicitly defined.

$$\frac{c \notin \mathbb{C} \quad M_c \rightarrow M_c' \quad \sigma' = \sigma[\mathbb{C}/\mathbb{C}; c]}{\begin{array}{c} \langle \textbf{capacity } id_c \ \{ \ M_c \ \}, \sigma \rangle \rightarrow \\ \langle \textbf{interface } id_c \textbf{ extends } \texttt{Capacity}\{ \ M_c' \ \}, \sigma' \rangle \end{array}} \tag{5}$$

$$\frac{\begin{array}{c} c \notin \mathbb{C} \quad S_c \subseteq \{id_x | x \in \mathbb{C}\} \quad id_c \notin superTypes(S_c) \\ M_c \rightarrow M_c' \quad \sigma' = \sigma[\mathbb{C}/\mathbb{C}; c] \end{array}}{\begin{array}{c} \langle \textbf{capacity } id_c \textbf{ extends } S_c \ \{ \ M_c \ \}, \sigma \rangle \rightarrow \\ \langle \textbf{interface } id_c \textbf{ extends } S_c \ \{ \ M_c' \ \}, \sigma' \rangle \end{array}} \tag{6}$$

3) Skill:

Definition 3: A skill s is a possible implementation of a capacity c_s fulfilling all the constraints of this specification. 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 7 and 8 describe the transformation 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[7]; and it is implicitly used as the extended type when this latest is not specified.

[6] `Capacity` in the SARL Library: http://www.sarl.io/docs/api/io/sarl/lang/core/Capacity.html

[7] `Skill` in the SARL Library: http://www.sarl.io/docs/api/io/sarl/lang/core/Skill.html

$$\frac{\begin{array}{c} s \notin \mathbb{S} \quad I_s \subseteq \{id_x | x \in \mathbb{C}\} \\ B = F_s \cup B_s \cup M_s \cup H_s \rightarrow B' \quad \sigma' = \sigma[\mathbb{S}/\mathbb{S}; s] \end{array}}{\begin{array}{c} \langle \textbf{ skill } id_s \textbf{ implements } I_s \ \{ \ B \ \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_s \textbf{ extends } \texttt{Skill} \textbf{ implements } I_s\{ \ B' \ \}, \sigma' \rangle \end{array}} \tag{7}$$

$$\frac{\begin{array}{c} s \notin \mathbb{S} \quad I_s \subseteq \{id_x | x \in \mathbb{C}\} \quad id_s \notin superTypes(S_s) \\ B = F_s \cup B_s \cup M_s \cup H_s \rightarrow B' \quad \sigma' = \sigma[\mathbb{S}/\mathbb{S}; s] \end{array}}{\begin{array}{c} \langle \textbf{ skill } id_s \textbf{ extends } S_s \textbf{ implements } I_s \ \{ \ B \ \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_s \textbf{ extends } S_s \textbf{ implements } I_s\{ \ B' \ \}, \sigma' \rangle \end{array}} \tag{8}$$

4) Agent:

Definition 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[8]. 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 9 and 10 provide the transformations for the **agent** construct.

$$\frac{\begin{array}{c} a \notin \mathbb{A} \quad B = F_a \cup B_a \cup M_a \cup H_a \rightarrow B' \\ \sigma' = \sigma[\mathbb{A}/\mathbb{A}; a] \end{array}}{\begin{array}{c} \langle \textbf{agent } id_a \ \{ \ B \ \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_a \textbf{ extends } \texttt{Agent}\{ \ B' \ \}, \sigma' \rangle \end{array}} \tag{9}$$

$$\frac{\begin{array}{c} a \notin \mathbb{A} \quad s_a \in \{id_x | x \in \mathbb{A}\} \quad id_a \notin superTypes(s_a) \\ B = F_a \cup B_a \cup M_a \cup H_a \rightarrow B' \quad \sigma' = \sigma[\mathbb{A}/\mathbb{A}; a] \end{array}}{\begin{array}{c} \langle \textbf{agent } id_a \textbf{ extends } s_a \ \{ \ B \ \}, \sigma \rangle \rightarrow \\ \langle \textbf{class } id_a \textbf{ extends } s_a \ \{ \ B' \ \}, \sigma' \rangle \end{array}} \tag{10}$$

IV. Example: Ping-Pong Application

In order to illustrate the transformations from the SARL language to the Java language, a simple application example is used. The system is composed by two types of agents. An `Initiator` agent sends a `Ping` event to an `Answerer` agent. When this latest agent receives the event, it sends a `Pong` event to the emitter of the received `Ping` agent. When the `Initiator` agent receives a `Pong` event, it writes a message to the computer console.

[8] `Agent` in the SARL Library: http://www.sarl.io/docs/api/io/sarl/lang/core/Agent.html

A. 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.

```
event Ping {
  var identifier : int
  new (id : int) {
    identifier = id
  }
}
event Pong {
  var identifier : int
  new (originalEvent : Ping) {
    identifier = id.identifier
  }
}
```

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

Regarding the `Ping` event, its identifier is defined as a 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 is the same as the identifier of the `Ping` event to which the `Pong` event corresponds. At Line 10, the identifier of the `Ping` event is copied into the `Pong` event.

The application of the transformation 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.

```
public class Ping extends Event {
  public int identifier;
  public Ping(final int id) {
    this.identifier = id;
  }
}
```

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

B. 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 exhibits to the agent the method to be invoked for writing a message (Line 2).

```
capacity Log {
  def showMessage(message : String)
}
skill ConsoleLog implements Log {
  def showMessage(message : String) {
    System::out.println(message)
  }
}
```

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 to the computer console (Line 5). The `System` type and the `println` method are provided by the Java library, which is accessible from any SARL program (Rodriguez et al., 2014).

Listing 4 provides the Java program that is equivalent to the SARL program in Listing 3.

```
public interface Log extends Capacity {
  void showMessage(String message);
}
public class ConsoleLog extends Skill implements Log {
  public void showMessage(String message) {
    System.out.println(message);
  }
}
```

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

C. Initiator Agent Definition

The behavior of the `Initiator` agent is divided into three parts. 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 for the calling agent. This function is invoked by the `Initiator` agent at Line 4 for registering the `ConsoleLog` skill to the `Initiator` agent. As soon as the `setSkill` is invoked, all the methods that are exhibited by the implemented capacity could be called from the agent's code. The access level of these exhibited methods from the code is enabled by the declaration of the **uses** statement at Line 2. This statement makes the methods from the specified capacities accessible from everywhere in the agent's code. Lines 6 and 9 illustrate two invocations at the capacities' methods: `emit` from the `DefaultContextInteractions` capacity, and `showMessage` from the `Log` capacity.

```
agent Initiator {
  uses DefaultContextInteractions, Log
  on Initialize {
    setSkill(new ConsoleLog)
    var pingEvent = new Ping(0)
    emit(pingEvent)
  }
  on Pong {
    showMessage("I have received the answer for event "
      + occurrence.identifier)
  }
}
```

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

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

The last part of the agent's behavior concerns the reception of the `Pong` event. This specific behavior is supported by the event handler, which is specified after the **on** keyword. The event handler is automatically invoked when the agent receives an occurrence of the `Pong` event. It writes a message through the `Log` capacity. The **occurrence** keyword represents the instance of the event for which the event handler is executed. The **occurrence**'s type is the same as the event type, which is specified after the **on** keyword.

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). 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 handler functions

are always added to the collection of the runnable event handler functions. 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.

```
public class Initiator extends Agent {
  private void $on$io_sarl_core_Initialize$0(
      final Initialize occurrence) {
    setSkill(new ConsoleLog());
    Ping pingEvent = new Ping(0);
    getSkill(DefaultContextInteractions.class).emit(
     pingEvent);
  }
  private void $on$Pong$0(final Pong occurrence) {
    getSkill(Log.class).showMessage(
      "I have received the answer for event "
      + occurrence.identifier);
  }
  public void $eval$io_sarl_core_Initialize(
       final Initialize occurrence, List<Runnable>
    handlers) {
     handlers.add(() >$on$io_sarl_core_Initialize$0(
    occurrence));
  }
  public void $eval$Pong(final Initialize occurrence,
       List<Runnable> handlers) {
    handlers.add(() >$on$Pong$0(occurrence));
  }
}
```

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 emit of the `Pong` event at Line 5 is scoped in order to restrict the receivers of the events to the source of the `Ping` event. The scoping expression is expressed with a lambda expression, which takes the address as the receiver as argument (the **it** keyword).

```
agent Answerer {
  uses DefaultContextInteractions
  on Ping {
    var pongEvent = new Pong(occurrence)
    emit(pongEvent) [it == occurrence.source]
  }
}
```

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.

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

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[9].

V. Related Works

Since Shoham, a multitude of APL 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 (Bordini et al., 2009) for a more comprehensive analysis.

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 (Rao and Georgeff, 1995) force agents to have a certain level of cognitive and reasoning capabilities, sometimes undesired in reactive MAS. From our point of view, BDI (Rao and Georgeff, 1995) 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 APL can be mentioned: SimpA (Ricci et al., 2011) and SimpAL (Ricci and Santi, 2011), 2APL (Dastani, 2008) and 3APL (Dastani et al., 2004), GOAL (de Boer et al., 2002) and Jason-AgentSpeak (Bordini et al., 2007). All of them provides a set of tools to support software application's development. SimpAL[10] provides an Eclipse-based IDE, including a compiler and a run-time support. The 2APL[11] 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- and event-driven entity. In SARL, agent's behaviors also communicate 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, 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 et al. (2017) proposed an agent-based domain specific language (DSL) upon the MARS framework. This Xtext-based DSL provides a set of features that intersects the ones provided by SARL. It is very interesting regarding the specification of reactive behaviors, and of probes on agents' attributes. We think that SARL provides a better support for hierarchical system implementation, and distribution support.

Within the architecture-independent APL, we may mention GAML agent-oriented language and its corresponding

[9]SARL Source code service: https://github.com/sarl/sarl/tree/master/contribs/io.sarl.examples/io.sarl.examples.plugin/projects

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

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

platform GAMA (Grignard et al., 2013). 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 share the same development approach based on Xtext.

VI. Conclusion and Perspectives

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 SARL run-time environment (SRE) provides the tools and the features that are mandatory for running such a program according to the SARL specifications. In this paper, we propose the SRE to be a Java-based application. We describe the transformation of the SARL constructs to equivalent Java's constructs.

Future work will focus on mapping well known agent architectures (e.g. BDI Rao and Georgeff (1995)) and organizational models (e.g. MOISE Hannoun et al. (2000) and CRIO Cossentino et al. (2010)). We believe that the corpus of concepts provided in SARL will allow us to map these models into SARL concepts.

References

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

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

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.

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

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, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, Eds. Springer Berlin Heidelberg, 2004, vol. 3067, pp. 111–130.

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

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

N. H. Gehani, H. V. Jagadish, and O. Shmueli, "Event specification in an active object-oriented database," *SIGMOD Rec.*, vol. 21, no. 2, pp. 81–90, Jun. 1992. [Online]. Available: http://doi.acm.org/10.1145/141484.130300

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

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

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 2013: Principles and Practice of Multi-Agent Systems*, ser. LNCS, G. Boella, E. Elkind, B. Savarimuthu, F. Dignum, and M. Purvis, Eds. Springer Berlin Heidelberg, 2013, vol. 8291, pp. 117–131.

M. Hannoun, O. Boissier, J. S. Sichman, and C. Sayettat, "MOISE: An organizational model for multi-agent systems," in *Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, M. C. Monard and J. S. Sichman, Eds., no. 1952. Springer Berlin Heidelberg, 2000, pp. 156–165.

J. Matheny, C. White, D. Anderson, and A. Schaeffer, "Object-oriented event notification system with listener registration of both interests and methods," Jul. 23 2002, uS Patent 6,424,354. [Online]. Available: https://www.google.com/patents/US6424354

P. Mosses, "Exploiting labels in structural operational semantics," *Fundam. Inform.*, vol. 60, no. 1–4, pp. 17–31, 2004.

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

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

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

A. Ricci, M. Viroli, and G. Piancastelli, "simpa: An agent-oriented approach for programming concurrent applications on top of java," *Sci. Comput. Program.*, vol. 76, no. 1, pp. 37–62, Jan. 2011.

S. Rodriguez, N. Gaud, and 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, Aug 2014, pp. 103–110.

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

C. Szyperski, J. Bosch, and W. Weck, *Component-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 184–192. [Online]. Available: http://dx.doi.org/10.1007/3-540-46589-8_10