

Agent Programming Language Review Assignment

SARL

Shane Cooke (17400206)

Overview

The agent programming language that I have decided to review is called SARL. SARL is a statically typed agent programming language that aims to provide the fundamental abstractions for dealing with concurrency, distribution, interaction, decentralisation, reactivity, autonomy, and dynamic reconfiguration. It has its roots in the Java programming language and unlike other JVM languages, has zero interoperability issues with Java. The SARL team are convinced that the agent-oriented paradigm in SARL is the key to the easy and practical implementation of modern complex software applications.

There are currently fifteen different developers cited as contributing to the SARL project, however the three original authors and creators of this project are Stéphane Galland, Nicolas Gaud and Sebastian Rodriguez. Galland and Gaud are based out of Burgundy Franche-Comté University in France, and Rodriguez is based out of RMIT University in Melbourne, Australia.^[1] They boast an array of different skills such as agent-based modelling, agent-based software development, simulation, and modelling, and each are professors in their fields of study. The other contributors on this project are from a very diverse array of countries such as Italy, the USA and Ireland^[2], and this goes to show that SARL is an international project.

SARL is an open source and completely free to use project for which the source code is available on the SARL GitHub page^[3]. There is an Eclipse SARL IDE, with SARL pre-installed and configured available on the official project website. There is a Windows, Mac, and Linux version and all are freely available for download. It is also supported by Maven, which makes it easily accessible through any modern Java IDE such as Eclipse or IntelliJ. Installation is very easy and detailed download instructions are available on the download section of the official SARL website.

SARL is an extremely dynamic language and can be used for a vast array of different problems. It can be used to create agent-based software, agent-based simulation software, holonic software and organizational software to name a few. The community page of the website shows a list of community made projects such as “Elevator Simulator”, “Human-Swarm Interaction”, “Agents in the Microsoft Airsim Framework” and “2D Environment Modelling” which also demonstrates just how vast an array of uses that the SARL language can accommodate^[4]. SARL has zero interoperability issues with the Java programming language, which makes it a great choice for Java Developers who wish to expand into the field of agent programming. SARL improves on many areas of Java such as agent specific statements, type inference, operator overloading and extension methods, and makes it a great choice for many modern complex software applications. SARL is platform-independent, agent’s architecture-agnostic and provides a set of agent-oriented first-class abstractions directly at the language level.

Agent Architecture Design

An *agent* is an autonomous entity that has a set of skills to realise the capacities it exhibits. To understand the architecture of an agent in SARL, it is important to understand the key components that make up the agent. Below is a figure of the architecture of an agent in SARL^[5], which I labelled with numbers for ease of reading in this review:

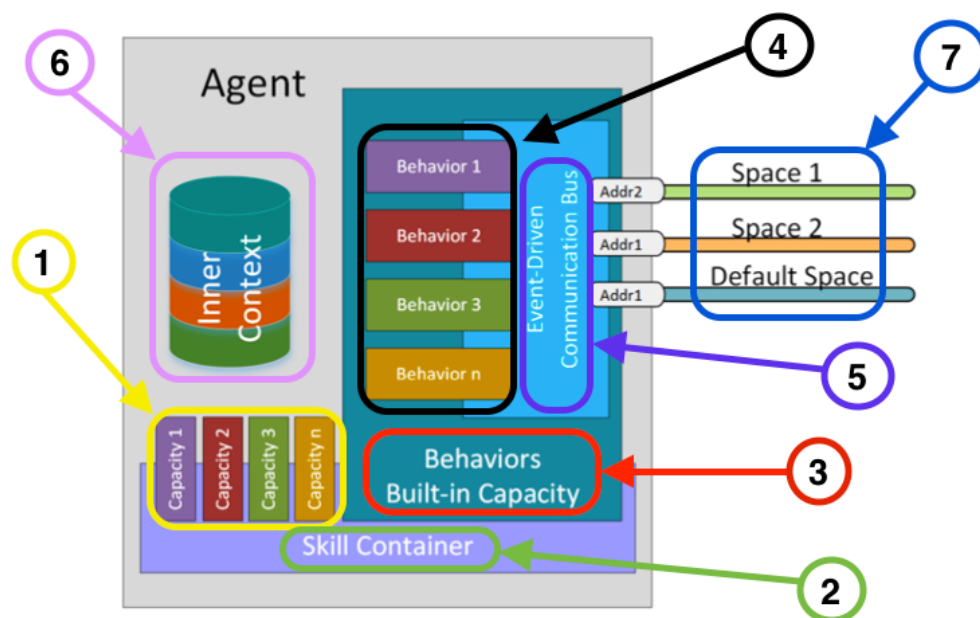


Fig. 2. An Agent in SARL

In SARL, an agent is an autonomous entity that has a set of skills to realise the capacities it exhibits. These *Capacities*⁽¹⁾ are the specification of a collection of actions. This specification makes no assumptions about its implementation and can be used to specify what an agent can do. Each agent also has a set of *Skills*⁽²⁾ to realise the capacities it exhibits. A skill is a possible implementation of a capacity fulfilling all of the constraints of the collection of actions which are specified by said capacity. These are stored in the “Skill Container” as illustrated in the figure above.

Each agent also has a set of *Built-in Capacities*⁽³⁾ that are essential to respect the commonly accepted competences of these agents. These built-in capacities are considered the main building blocks on top of which other higher-level capacities and skills can be constructed. The built-in capacities of each agent are the behaviour capacities that enable agents to incorporate a collection of behaviours that will determine its global conduct. These *Behaviours*⁽⁵⁾ are in the same space as the built-in capacities and map a collection of perceptions to a sequence of actions.

The collection of perceptions mapped by behaviours to a sequence of actions are represented by *Events*⁽⁵⁾. The various behaviours of an agent communicate using an event-driven approach. These *events* are defined as the specification of some occurrences in a space that may potentially trigger effects by a listener and are communicated through the “Event-Driven Communication Bus” as illustrated in the figure above.

A *Context*⁽⁶⁾ defines the perimeter or boundary of a sub-system, and gathers a collection of spaces. In each context, there is at least one particular space called the default space to which all agents in the context belong. This ensures the existence of a common shared

space to all agents in the same context, and each agent can then create specific public or private spaces to achieve its personal goals. 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.

A Space⁽⁷⁾ 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.

Now that we understand the underlying components of an agent, we can summarise the whole agent architecture. Each agent in SARL contains capacities which are the specification of the actions the agent wishes to carry out. These actions have constraints on them which are specified by the capacity to which they belong. Each agent also has a set of skills, which are possible implementations of the capacities of the agent that fulfil all of these constraints. Each capacity can be realised by these skills, and they are available to the agent when the “Initialise” event is called. Each agent also has a behaviour capacity, that allows an agent to incorporate a collection of behaviours that will determine its global conduct. These behaviours map a set of perceptions to a sequence of actions contained in the capacities. These perceptions are represented as events, which are then relayed into a space via the event-driven communication bus to potentially trigger effects by listeners who are waiting for said event to be executed. These events carry the information necessary for multi-agent systems and allow communication between agents.

Agents in SARL use a mental state of actions and behaviours. The actions an agent can perform can be specified by the capacities of an agent or natively inside the agent’s body. The mental state of an agent is implemented as a collection of attributes in the agent and can be defined as follows:

```
agent TestAgent {  
    var mentalStateElement1 : String  
    var mentalStateElement2 : int  
}
```

The behaviours an agent can exhibit are specified in the behaviours section of the agent architecture. The attributes that make up the mental state will be used by the behaviours of the agent to carry out any calculations or processes that the agent was built for.

SARL does not impose a specific agent’s control loop. When agents are created by the host VM or another agent, the host VM is in charge of creating the agent instance and installing the skills associated to the built-in capacities of the agent. Then when the agent is ready to begin its execution, it fires the *Initialise* event. This event contains any parameters for the agent’s instance. Likewise, when the agent has decided to stop its own execution, the VM will fire a *Destroy* event to enable the agent to release any resources it may still hold. 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.

How to Code an Agent

In this section, I will outline the key steps to coding a basic agent in SARL.

The first step of coding an agent in SARL is being able to define an agent. To do this, the *agent* operator is used along with the desired name of the agent:

```
agent TestAgent {  
}
```

Next, we need to define the imports necessary for our agent to use. SARL uses the standard Java paradigm of *import X*, and inside the agent, the *uses* command is necessary to specify which imports the agent is using:

```
import io.sarl.core.Lifecycle  
  
agent TestAgent {  
    uses Lifecycle  
}
```

Next, we must define the mental state of the agent. The mental state of an agent in SARL is composed by the data in the agent and is implemented as a collection of attributes. The *var* keyword is used to define attributes in SARL:

```
agent TestAgent {  
    var mentalStateElement1 : String  
    var mentalStateElement2 : int  
}
```

The next step of programming an agent is handling events. Every agent follows the *Initialize* (start-up), *Lifetime*(lifecycle) and *Destroy*(ending) events, which are vital for programming agents in SARL. To do so, we use the *on* command with the desired event, and specify what we would like the agent to do upon said event:

```
agent TestAgent {  
    uses Lifecycle, Logging  
  
    on Initialize {  
        info("I am ready!")  
    }  
  
    on Destroy {  
        info("I am finished!")  
    }  
}
```

The next step is to give our agent an action to carry out. To do this we need to declare a function in our agent by using the *def* operator. The *def* operator requires a function name, a set of parameters, a return type, and a body in the form:

```
def Function1(a : int, b : int): int {  
    return a + b  
}
```

To call this function, we simply use:

```
Function1(5,4)
```

Another very important aspect of coding agents in SARL is the *in* operator, which specifies a time for the agent to wait to execute something. In the case below, the agent will wait 2000 milliseconds (2 seconds) and then use the *killMe* command to kill itself and end processes:

```
in(2000)[killMe]
```

Now that we understand some of the basics of coding an agent in SARL, we will use all of the previously discussed elements to create a basic agent that adds two numbers and returns the answer in the terminal. The code for the following is as follows:

```
import io.sarl.core.Destroy
import io.sarl.core.Initialize
import io.sarl.core.Lifecycle
import io.sarl.core.Logging
import io.sarl.core.Schedules

agent TestAgent {
    uses Lifecycle, Logging, Schedules

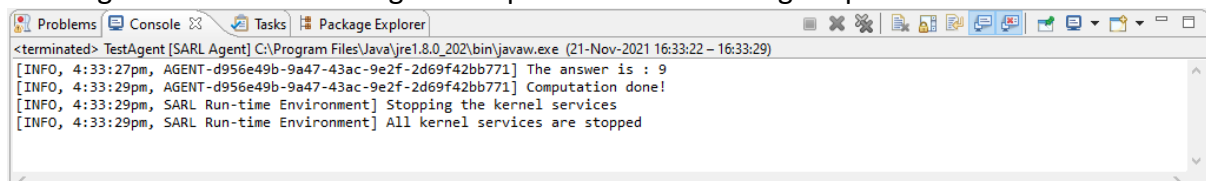
    var a : int
    var b : int

    def Function1(a : int, b : int): int {
        return a + b
    }

    on Initialize {
        info("The answer is : " + Function1(5,4))
        in(2000)[killMe]
    }

    on Destroy {
        info("Computation done!")
    }
}
```

Running this code as a SARL agent will produce the following output:



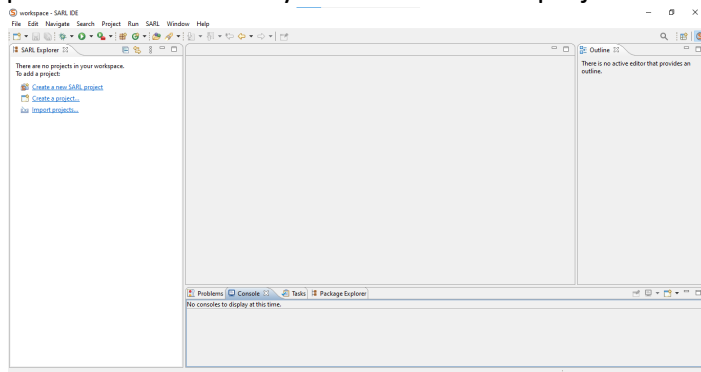
Above, I have outlined some of the key aspects to begin coding agents in SARL. As with all programming languages, there are more complex elements such as extensions, inheritance, skills, getter methods and reactive behaviours. These elements are of course important in their own regard, however for the purpose of this report, I have only explained the vital elements which you need to begin coding agents in the SARL programming language.

Example Application

Below I will give a detailed description of how to write your first “Hello World” application in the SARL agent programming language.

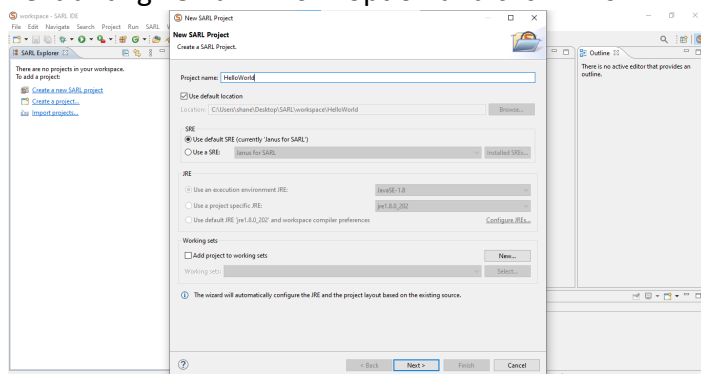
Step 1: Downloading and Launching Eclipse SARL

There are many ways to run SARL on your computer, however I found that the best way was to download the pre-configured Eclipse with SARL SDK from the SARL website. Upon downloading and running this file, an Eclipse instance will open with all of the configurations pre-loaded to create your “Hello World” project:



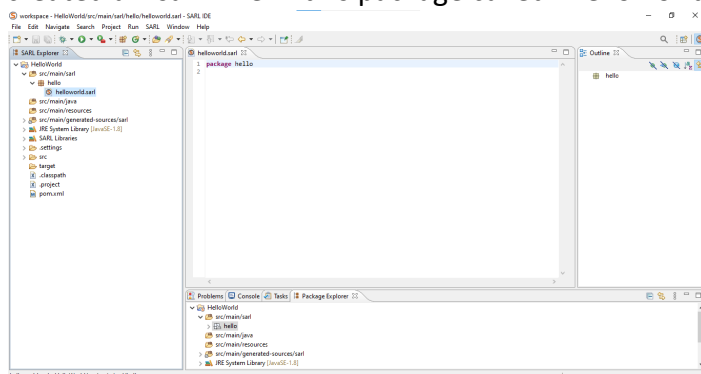
Step 2: Creating a project

Next, click “Create a new SARL project” which will launch a window prompting you to give a project name and the configurations you desire. When you have your desired choices, click next until given a “Finish” option and click Finish.



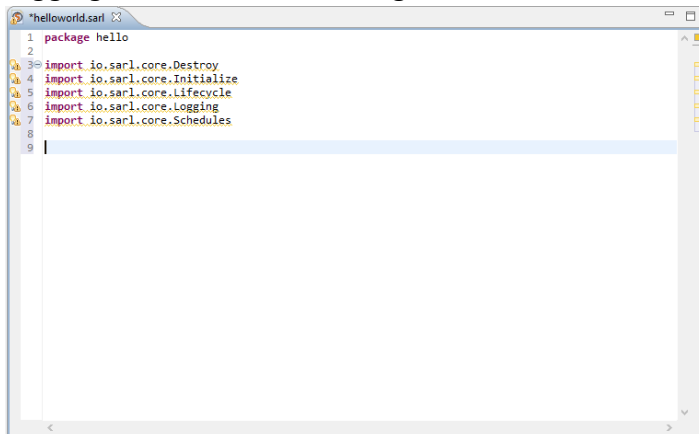
Step 3: Create a SARL file

Once you have your blank SARL project, the next step is to create a SARL file. The SARL file should be created in the “src/main/sarl” folder. I created a package called “hello” and then created a “.sarl” file in this package called “helloworld.sarl” as illustrated below:



Step 4: Add necessary imports

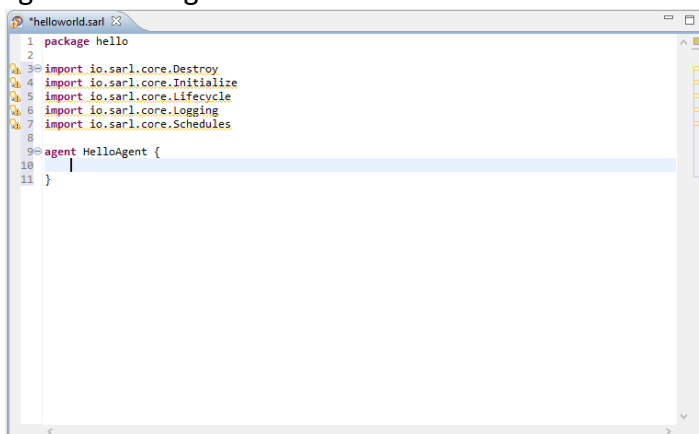
The next step of this process is to add the necessary imports for a helloworld agent. I added the *Destroy*, *Initialize*, *Lifecycle*, *Logging* and *Schedules* imports. *Destroy* will be used to kill the agent, *Initialize* will be used to start the agent, *Lifecycle* is necessary to kill a process, *Logging* is used to name the agent and *Schedules* is used to time the agent.



```
1 package hello
2
3 import io.sarl.core.Destroy
4 import io.sarl.core.Initialize
5 import io.sarl.core.Lifecycle
6 import io.sarl.core.Logging
7 import io.sarl.core.Schedules
8
9
```

Step 5: Create the agent

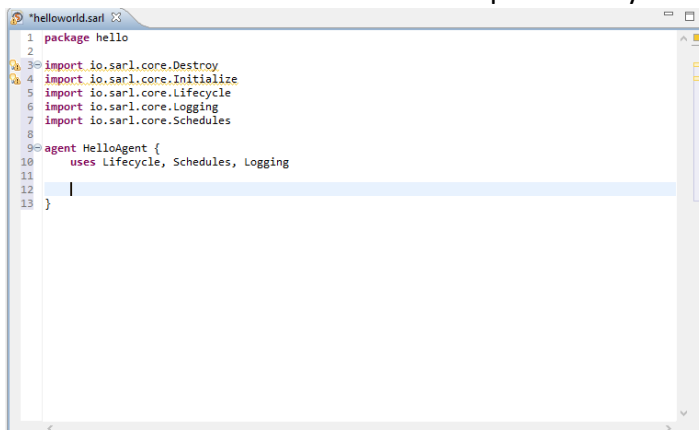
The next step is to create the agent. To do this use the *agent* command. I decided to call my agent "HelloAgent" and the code for this is illustrated below:



```
1 package hello
2
3 import io.sarl.core.Destroy
4 import io.sarl.core.Initialize
5 import io.sarl.core.Lifecycle
6 import io.sarl.core.Logging
7 import io.sarl.core.Schedules
8
9 agent HelloAgent {
10
11 }
```

Step 6: Specify imports for agent

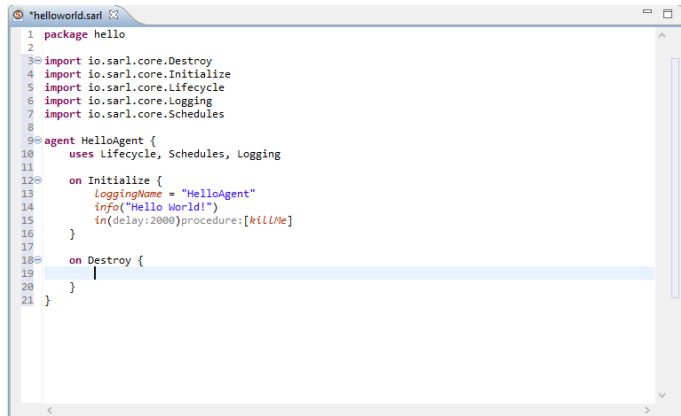
Next, you will specify what imports that your agent is going to use. To do this you use the *uses* command and indicate which imports that your agent will need:



```
1 package hello
2
3 import io.sarl.core.Destroy
4 import io.sarl.core.Initialize
5 import io.sarl.core.Lifecycle
6 import io.sarl.core.Logging
7 import io.sarl.core.Schedules
8
9 agent HelloAgent {
10     uses Lifecycle, Schedules, Logging
11
12 }
13
```

Step 7: Specify what the agent will do when Initialized

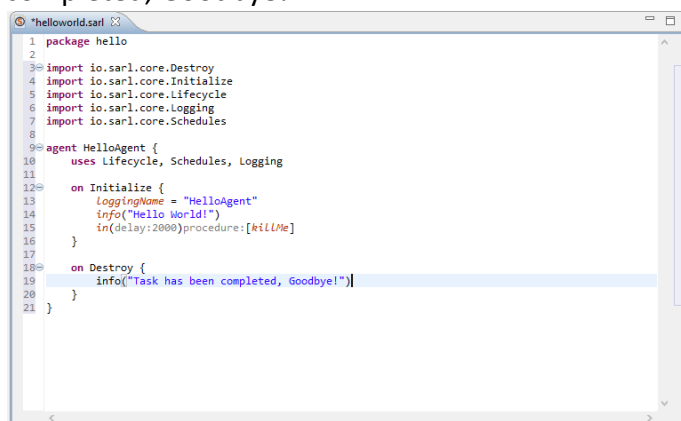
Next, you need to tell your agent what to do when it is initialized. To do this you use the *on Initialize* command and fill the body with what you would like the agent to do. In this case, I use the *info* command to print to the terminal, and I use the *in* command to tell the agent to wait two seconds before killing itself using the *killMe* command:



```
1 package hello
2
3 import io.sarl.core.Destroy
4 import io.sarl.core.Initialize
5 import io.sarl.core.Lifecycle
6 import io.sarl.core.Logging
7 import io.sarl.core.Schedules
8
9 agent HelloAgent {
10     uses Lifecycle, Schedules, Logging
11
12     on Initialize {
13         loggingName = "HelloAgent"
14         info("Hello World!")
15         in(delay:2000)procedure:[killMe]
16     }
17
18     on Destroy {
19     }
20 }
21 }
```

Step 8: Specify what the agent will do when Destroyed

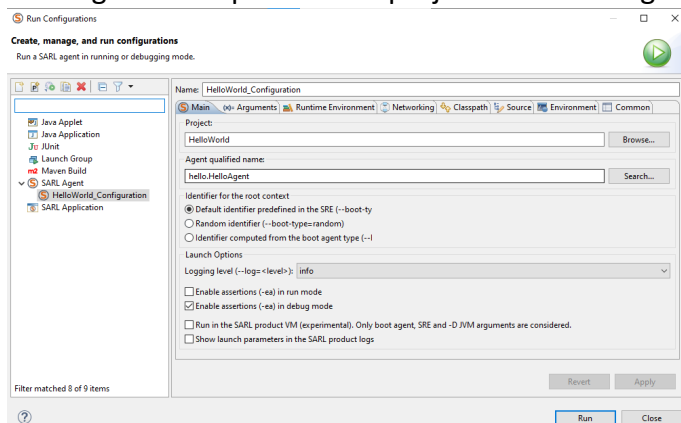
Now, you need to tell your agent what to do when it receives the *Destroy* event. I have decided to print to terminal using the *info* command with a message of “Task has been completed, Goodbye!”:



```
1 package hello
2
3 import io.sarl.core.Destroy
4 import io.sarl.core.Initialize
5 import io.sarl.core.Lifecycle
6 import io.sarl.core.Logging
7 import io.sarl.core.Schedules
8
9 agent HelloAgent {
10     uses Lifecycle, Schedules, Logging
11
12     on Initialize {
13         loggingName = "HelloAgent"
14         info("Hello World!")
15         in(delay:2000)procedure:[killMe]
16     }
17
18     on Destroy {
19         info("Task has been completed, Goodbye!")
20     }
21 }
```

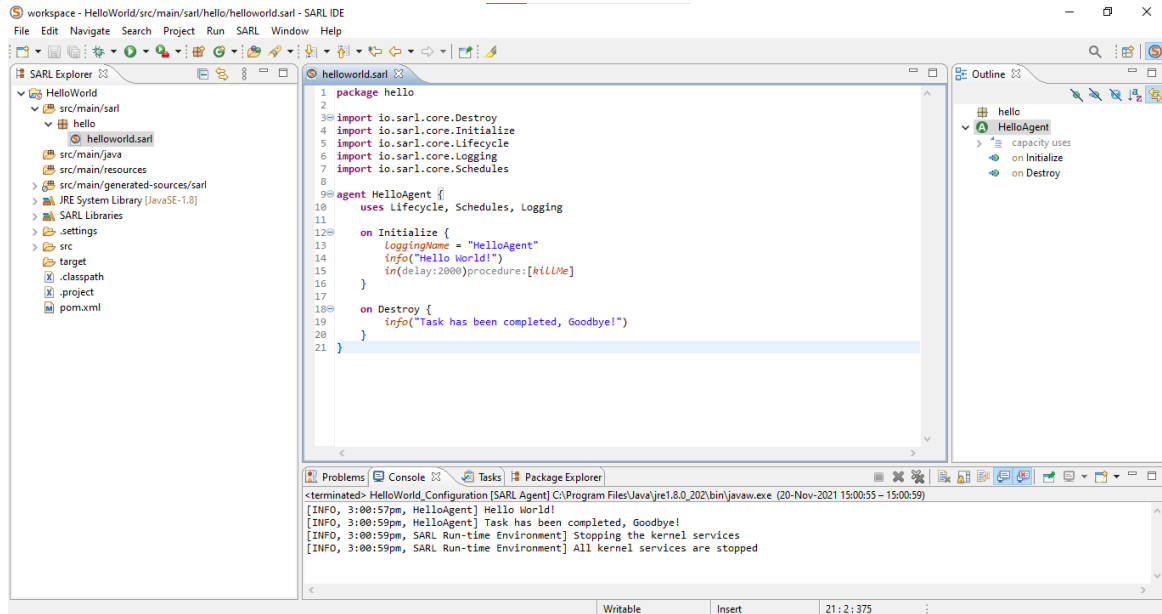
Step 9: Click run and choose run configuration

Once this is completed, you can run your agent. To do this, click the green play button in the SARL Eclipse IDE, and a configuration window will pop up. Create a new configuration of SARL agent that specifies the project name and agent name. When satisfied, click “Run”:



Step 10: Output

Once run is clicked, if you have written the code properly and chosen the correct run configuration, you will see an output. As you can see, when the agent is initialized “Hello World” is outputted to the terminal, and when the agent is destroyed “Task completed, Goodbye!” is outputted, meaning that our HelloAgent was successful.



Conclusions

In conclusion, I really enjoyed my experience with SARL as a programming language. There is lots of documentation available on the official SARL website which goes into great detail about all aspects of this programming language. The agent architecture was quite complex and hard to understand at first, however after some thorough study on the subject, it became much more clear to me how agents worked. The documentation for coding agents is very thorough and all areas are covered. I think SARL is a very useful and practical Agent Programming Language, and the diverse array of projects in the community section of the official website illustrate this perfectly.

While I had an overall very positive experience with SARL, there are a few minor issues that I had while reviewing this language. The first of which is that SARL is not fully compatible with the latest version of the Eclipse IDE, which was giving me some unexpected errors and took me quite long to troubleshoot. The Eclipse SARL IDE was also not compatible with my Apple Mac M1 laptop, which meant that I had to use a windows laptop to complete this review. I had to download an earlier version of the Eclipse IDE (2020-06) and set up my configurations properly to get SARL working properly. The second minor problem I found with SARL is that a lot of the documentation is written in quite poor English. The documentation is vast and covers all possible subjects, however as a lot of the contributors to SARL are based in France, Italy, and other non-English speaking countries, I'm guessing the people who wrote the documentation may not speak English as their first language. Some of the grammatical errors and misuse of words sometimes made it hard to understand the documentation, especially on the topic of agent architecture. This was quite frustrating at times, however I would not class it as a major issue, and upon thorough study all aspects of SARL were understandable.

These relatively minor issues aside, I found programming in SARL an overwhelmingly positive experience. Once I had the Eclipse SARL IDE up and running, programming agents was simple and very practical. The documentation is overall very good apart from the grammatical issues, and there is lots of information available about agent architecture, installing SARL, programming agents, run configurations and SARL agent interaction. SARL is a practical and powerful agent programming language and I thoroughly enjoyed studying the architecture and coding agents in SARL.

References

- [1] Team, S.A.R.L., SARL. *About SARL*.
Available at: <http://www.sarl.io/about/index.html>
[Accessed November 18, 2021].
- [2] Sarl, Contributors to SARL/SARL. GitHub.
Available at: <https://github.com/sarl/sarl/graphs/contributors>
[Accessed November 21, 2021].
- [3] Sarl, SARL/SARL: SARL agent-oriented programming language.
Available at: <https://github.com/sarl/sarl>
[Accessed November 21, 2021].
- [4] Team, S.A.R.L., SARL community. Available at:
<http://www.sarl.io/community/index.html>
[Accessed November 21, 2021].
- [5] Rodriguez, S., Gaud, N., Galland, S., **SARL: a general-purpose agent-oriented programming language**.
Available at:
https://www.researchgate.net/publication/267759078_SARL_A_General-Purpose_Agent-Oriented_Programming_Language
[Accessed November 21, 2021].