ARIA LANGUAGE
Towards Agent Orientation Paradigm

Mohsen Lesani and Niloufar Montazeri
Computer Science Department, Shahid Bahonar University of Kerman, Kerman, Iran

Keywords: Agent Oriented Software Engineering, Agent Oriented Language, Software Agents.

Abstract: Agent oriented paradigm is one of the new contributions to the field of software engineering and has the potential to significantly improve current practice of the field. The paradigm should be elaborated both practically and conceptually. The contribution of this paper is twofold. Firstly, an agent oriented language called Aria and its compiler are proposed. Aria language is a superset of Java language and the compiler compiles a program in Aria to an equivalent program in Java. This enables Aria to fully integrate with and preserve all the existing knowledge and code in Java. Secondly, the three well-known object oriented principles of abstraction, inheritance and polymorphism are redefined for agent orientation. Two sample cases are presented: a chat room and an agent oriented MVC pattern.

1 INTRODUCTION

Objects are passive entities and the thread of execution is not a primary concept in object orientation. On the other hand, increasing multi core architectures provide hardware support for concurrency. In a sheer agent oriented approach, a system is designed as a team of cooperating autonomous agents. The behavior of a multi-agent software system is the emergence of cooperation of the agents. In contrast to an object that is a passive entity, an agent is modeled as a proactive social entity. An object does not change state or make any other object to change state unless it is told through methods to do so. On the contrary, an agent is alive and its behaviors are autonomously effective regardless of other agents.

The developer's attitude to shift to the new language and hence the success of language is highly dependent on backward compatibility i.e. preserving the developer's existing knowledge and code. While there are some agent oriented languages available, few of them are backward compatible with object oriented languages. More importantly, while object orientation is known to have three principles, publications from neither of the previous languages have theoretical discussions of the new paradigm.

2 ARIA AGENT ORIENTATION

Aria agent oriented language proposes language built-in support for specification of autonomous agents. It conceptually overrides and extends the three well known object orientation principles.
1. Abstraction: Everything is an agent or an object. Agents interact by sending messages to each other.
2. Inheritance: An agent can inherit message servicing and behaviors from a parent agent.
3. Polymorphism: An agent can override its parent definitions for message servicing and behaviors and the overriding definitions are effective even when the agent is referenced as of its parent type.

3 ARIA LANGUAGE

3.1 Abstraction

An agent is specified in Aria as syntax in Code Snippet 1. An agent is abstracted to perceive messages of definite types and have several concurrent behaviors.

Aria proposes a superior abstraction of messaging concept in comparison to object orientation. The object orientation abstraction for messaging is to call methods on objects that is conceptually a blocking message passing mechanism. Inter-agent communication is possible through not only blocking but also non-blocking and polling mechanisms. An agent can send messages to other agents in three different ways that are through tell, tellAndWait and tellAndPoll message passing methods. An agent can tell a message and proceed with its current behavior. An agent can also tellAndWait a message and block for its reply before advancing. The third approach is to tellAndPoll a message and iteratively check for the reply. This approach called polling allows for situations when a behavior should be sustained while a message reply is also waited for. These approaches are syntactically specified as in Code Snippet 2, Code Snippet 3 and Code Snippet 4.

An agent may receive messages of different class types. The perceive block for a definite message class is where every received message that is an instance of that class is directed to. Every agent has a hidden thread-safe message queue. All the messages that other agents send to the agent are put into its message queue. When an agent specification is compiled, a hidden message dispatching behavior is added to the agent behaviors. The message dispatching behavior continuously iterates the message queue, identifies the type of each message and runs the perceive block of the identified message type with the message as the parameter. All the message queuing and runtime type identification issues are handled behind the scene by the code that is generated and inserted by the compiler into the user code and in part by classes of Aria core package. As all the perceive blocks of an agent are executed sequentially in a single thread, a perceive block can only contain a short processing on the message. For instance it can contain the action that a simple reflex agent performs in realizing a message of a definite type. This is most common for user interface agents. As most of the messages they receive are requests for presentation and such requests can be carried out rapidly, user interface agents are usually reflex agents.

Behaviors are where the agent's processing should be coded. The code snippet of a behavior is translated to a repeatedly running code. Every behavior is executed on a separate thread by default and this supports concurrent behaviors in an agent.

Processing needed to reply some message types may be time consuming and messages of such types can not be promptly answered. As perceive blocks should only contain short processing tasks, there is a need that messages of time consuming types be directed to a behavior to be further processed. This could be coded in Aria as coded in Code Snippet 5. To support the user to accomplish this much easier, Aria allows defining message processing behaviors. A behavior can declare to process a definite message type and the presented code is automatically generated by the compiler. This means that Code Snippet 6 has exactly the same effect as Code Snippet 5.

A message that is being processed either in a perceive or behavior block can be replied by the reply keyword as shown in Code Snippet 7. When a message is replied, if the sender is waiting for the reply, the sender unblocks and gets the reply message as the return value of tellAndWait method. But if the sender is not waiting for the reply, the reply is simply sent to it to be queued and processed later. Reply statements without an explicit message are usually used to unblock sender agents that are waiting for a task to be finished. All the needed synchronizations are handled by Aria core package.

An agent is created and made alive as shown in Code Snippet 9. Agent's (hidden) message dispatching behavior and all the behaviors in the agent definition are started when the agent is commanded to become live. The atBirth block is executed when the agent is becoming live just before any of the behaviors are started. An agent can be requested to terminate by telling it a message of TerminateRequestMessage class. When a message of TerminateRequestMessage class is received, the agent dies by default by terminating all its behaviors and executing the atDeath block when all the behaviors are terminated. This default reaction to TerminateRequestMessage can also be overridden easily by providing a perceive block for it.

Agent definitions support all the constructs that can be defined inside class definitions. Fields can be defined for agent information storage. As fields are accessible from all the perceive and behavior blocks and agent behaviors can execute concurrently, care should be taken for synchronizing field access. Methods can also be defined for an agent but rarely an agent's method has a public access. Composition serves as a way to reuse existing agents and built more high-level agents with greater capabilities from them. An agent can obviously be composed of other agents. Each subagent can be capable of performing a part of the agent's responsibilities. The agent itself can act as a manager or coordinator.

3.2 Inheritance and Polymorphism

Aria agent specification supports inheritance in both agent specialization and service provision.

3.2.1 Agent Specialization

In addition to agent composition, agent specialization can be employed to achieve software reuse principle. All the capabilities present in an existing agent type can be reused by specializing a new agent from it and then the new agent can be supplemented with further capabilities. Agent specialization is the counterpart to class inheritance. While object oriented inheritance involves fields and methods, agent orientated specialization also concerns message processing mechanisms and behaviors. When a child agent inherits from a parent agent, the entire parent's perceive and behavior blocks are inherited by the child agent. The child agent can perceive all the message types that its parent could perceive and has all the behaviors that its parent has. In addition, new specific functionalities can be added.

3.2.2 Service Provision

A service is specified in Aria in the syntax shown in Code Snippet 8. A service specification formally defines a service that agents may provide. A service specification declares some message or request types. An agent that declares to support a service should be able to perceive all the message types declared in the service. An agent that has declared the perceive block for a message type is able to perceive messages of that type. An agent that has a message processing behavior for a definite message type is also considered to be able to perceive messages of that type. This is because the compiler automatically generates a perceive block for a message processing behavior. Service provision is the counterpart of interface realization in object orientation. A service can declare to extend other services. An agent that declares to service a definite Service B that extends another Service A, should be able not only to perceive all the messages declared in Service B but also all the messages declared in A. An agent can offer different services. Different agents can provide a unique service with different implementations.

3.2.3 Polymorphism

A general agent with definite capabilities can be specialized to have the capabilities more specifically defined. A perceive or behavior block can be overridden by an inheriting agent. A behavior defined in a child agent that has the same name as of a behavior in its parent agent overrides the parent's behavior. A perceive block defined in a child agent for a specific message class overrides the perceive block for the same message class in the parent agent. Sending a message to an upcasted child agent is polymorphic. This means that the perceive block defined in the child agent specification is executed rather than the perceive block defined in the parent agent specification.

An agent can add a perceive or behavior block to itself at runtime. The added perceive or behavior block can be a new or an overriding one. Hence, an agent can not only override the perceive and behavior blocks of its ancestors at compile time, but it can also override inherited and even its own perceive and behavior blocks at runtime. This supports an agent to change its behaviors in the course of its life as a result of learning or adaptation. As an instance, emerging a new message processing behavior block can be performed at runtime as coded in Code Snippet 10.

4 ARIA COMPILER (ARIAC)

Aria compiler is developed employing Antlr v.3 tool. Ariac translates a program in Aria language to a semantically equivalent program in Java language that is then compiled to Java bytecode. Besides agent and service specifications, Ariac compiler also accepts all the Java language constructs. It means that Aria language is a superset of Java language and Aria code is fully integrable with Java code.

The compiler has successfully passed compiling two sample cases implemented in Aria. please check http://ce.sharif.edu/~mohsen_lesani/aria.htm for more information.

REFERENCES

Bellifemine, F., Bergenti, F., Caire, G., & Poggi, A. (2005). JADE - A Java Agent Development Framework. In R. H. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni, *MultiAgent Programming: Languages, Platforms and Applications.* Springer-Verlag.

Howden, N., Ronnquist, R., Hodgson, A., & Lucas, A. (2001). JACK - Summary of an Agent Infrastructure. *5th International Conference on Autonomous Agents.*

Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). JADEX: A BDI Reasoning Engine. In R. H. Bordini, M. Dastani, J. Dix, & A. El Fallah Seghrouchni, *MultiAgent Programming: Languages, Platforms and Applications.* Springer-Verlag.

```
    public agent AgentType specializes parentAgentType services ServiceType1, ServiceType2, ... {
         atBirth {
         }
         perceive(Massageclass1 message) {
         }
         behavior behaviorName {
         }
         behavior behaviorName processes (MessageClass2 message) {
         }
         atDeath {
         }
         // Any OO field or method is also supported.
    }
```

Code Snippet 1

```
    agentName.tell(messageName)
```

Code Snippet 2

```
    Message message = agentName.tellAndWait(messageName)
```

Code Snippet 3

```
    MessageWaitedFor messageWaitedfor = agentName.tellAndPoll(messageName)
    while (!messageWaitedFor.isMessageReplied()) {
         // Do some tasks
    }
    Message replyMessage = messageWaitedFor.getMessage();
```

Code Snippet 4

```
    private ThreadSafeQueue<MessageClassType> ariaMessageClassTypeQueue =
         new ThreadSafeQueue<MessageClassType>();

    perceive(MessageClass message) {
         ariaMessageClassQueue.add(message);
    }
    behavior behaviorName {
         try {
          MessageClass message = ariaMessageClassQueue.remove();
          // behavior code to process message
         } catch (Exception e) {
          idle();
         }
    }
```

Code Snippet 5

```
    behavior behaviorName processes (MessageClassType message) {
         // behavior code to process message
    }
```

Code Snippet 6

```
    reply message
```

or

```
    reply
```

Code Snippet 7

```
AgentType agentName =  new AgentType();
agentname.becomeLive();
```

Code Snippet 9

```
    public service ServiceType
     extends AnotherServiceType1,
              AnotherServiceType2,
              ...
    {
         servicesTo(MessageClass1);
         servicesTo(MessageClass2);
    }
```

Code Snippet 8

```
addBehavior(
     new MessageProcessingBehavior<MessageClass>("BehaviorName")  {
          public void behavior(MessageClass message) {
               //Message Processing behavior code
          }
     }
);
```

Code Snippet 10