Agent-Oriented Computing: Agents as a Paradigm for Computer Programming and Software Development

Alessandro Ricci
DEIS, Alma Mater Studiorum – Università di Bologna
Via Venezia 52, 47521 Cesena (FC), Italy
a.ricci@unibo.it

Andrea Santi
DEIS, Alma Mater Studiorum – Università di Bologna
Via Venezia 52, 47521 Cesena (FC), Italy
a.santi@unibo.it

***Abstract*—The notion of *agent* more and more appears in different contexts of computer science, often with different meanings. The main acceptation is the AI (Artificial Intelligence) and Distributed AI one, where agents are essentially exploited as a technique to develop special-purpose systems exhibiting some kind of intelligent behavior. In this paper, we introduce a further perspective, shifting the focus from AI to computer programming and programming languages. In particular, we consider agents and related concepts as general-purpose abstractions useful for programming software systems in general, conceptually extending object-oriented programming with features that – we argue – are effective to tackle some main challenges of modern software development. Accordingly, the main contribution of the work is first the definition of a conceptual space framing the basic features that characterize the agent-oriented approach as a programming paradigm, then its validation in practice by using a platform called JaCa, with real-word programming examples.**

***Keywords*-agent-oriented programming; multi-agent systems; concurrent programming; distributed programming**

I. Introduction

The notion of *agent* more and more appears in different contexts of computer science, often with different meanings. In the context of Artificial Intelligence (AI) or Distributed AI, agents and multi-agent systems are typically exploited as a technique to tackle complex problems and develop intelligent software systems [16][32][27]. In this paper, we discuss a further perspective, which aims at exploiting agents and agent-oriented abstractions to devise a high-level computing programming paradigm for developing software, as a natural evolution of objects (as defined in OOP) and actors [6]. So, instead of exploiting agents as abstractions to support AI techniques, here we frame the value of multi-agent programming as a general-purpose paradigm for organizing and programming software, providing features that we consider effective to tackle main challenges of modern and future software development, such as concurrency, decentralization of control, distribution, autonomy, adaptivity.

Concurrency, in particular, due to the spread of multi-core technologies, is more and more a core issue of mainstream programming—besides the academic research contexts where it has been studied for the last fifty years. This situation is pretty well summarized by the sentence: "The free lunch is over" as put by Sutter and Larus in [30]. Besides introducing fine-grain mechanisms or patterns to exploit parallel hardware and improve the efficiency of programs in existing mainstream languages, it is now increasingly important to introduce higher-level abstractions that "help build concurrent programs, just as object-oriented abstractions help build large component-based programs" [30]. We argue that agent-oriented programming – as framed in this paper – provides one such level of abstraction. Besides concurrency, we believe that the level of abstraction introduced by an agent-oriented programming paradigm would be effective to tackle the complexities introduced by modern and future application domains, such as cloud computing, autonomic computing, pervasive computing and so on.

Actually, the idea of Agent-Oriented Programming is not new. The first paper about AOP is dated 1993 [28], and since then many Agent Programming Languages (APL) and languages for Multi-Agent Programming have been proposed in literature [11][12][13]. The objective of AOP as introduced in [28] was the definition of a post-OO programming paradigm for developing complex applications, providing higher-level features compared to existing paradigms. In spite of this objective, it is apparent that agent-oriented programming has not had a significant impact on mainstream research in programming languages and software development, so far. We argue that this depends on the fact that (in spite of few exceptions) most of the effort and emphasis have been put on theoretical issues related to AI themes, instead of focusing on the key principles and practice of general-purpose computer programming. This is the direction that we aim at exploring in our work and in this paper.

The remainder of the paper is organized as follows. After presenting related works (Section II), we first define a conceptual space to describe the basic features of a general-purpose programming paradigm based on agent-oriented abstractions (Section III). Then, we provide a first practical evaluation by exploiting an agent-oriented platform called JaCa (Section IV), which actually integrates two different existing agent technologies, Jason [9][10] and

CArtAgO [25]. The objective is to show how to exploit agent-oriented abstractions to conceive and develop real-world programs, and point out outcomes and limitations of current models and technology. Finally we close the paper with some concluding remarks (Section V).

II. Related Works

Most of the agent-oriented programming languages and technologies – in particular those based on high-level computational model/architecture such as the BDI (Belief-Desire-Intention) one [23] – have been introduced in (Distributed) Artificial Intelligence, so targeted to problems in that context [11][12][13]. Besides this main perspective, in the context of AOSE (Agent Oriented Software Engineering) some agent-oriented *frameworks* based on mainstream programming languages – such as Java – have been introduced, targeted to the development of complex distributed software systems. A main example is JADE (Java Agent DEvelopment Framework) [8], a FIPA-compliant [1] platform that makes it possible to implement multi-agent systems in Java. JADE is based on a weak notion of agency: JADE agents are Java-based actor-like active entities, communicating by exchanging messages based on FIPA ACL (Agent Communication Language). So there is not an explicit account for high-level agent concepts – goals, beliefs, plans, intentions are examples, referring to the BDI model – that are exploited instead in agent-oriented programming languages to raise the level of abstraction adopted to define agent behaviour. Also, JADE has not an explicit notion of agent environment, defining agent actions and perceptions, which are key concepts for defining agent reactiveness. Differently from JADE, the JaCa platform presented in this paper allows for programming agents using a BDI-based computational model and has explicit notion of shared programmable environments – perceived and acted upon by agents – based on the A&A (Agents and Artifacts) conceptual model [20], described in next sections.

Another example of Java-based agent-oriented framework is simpA [26], which has been conceived to investigate the use of agent-oriented abstractions for simplifying the development of concurrent applications. simpA shares many points with the perspective depicted in this paper: however it is based in on a weak model of agency, similar to the one adopted in JADE. Differently from JADE, it explicitly supports a notion of environment, based on A&A.

Besides the different underlying computational models, both JADE and simpA do not explicitly introduce a new full-fledge agent-oriented programming language for programming agents, being still based on Java. A different approach is adopted by JACK [15], a further platform for developing agent-based software which *extends* the Java language with BDI constructs – such as goals and plans – for programming agents, integrating the object-oriented and agent-oriented levels. Finally, similarly to JADE, Jadex [22] is a FIPA compliant framework based on Java and XML, but adopting the BDI as underlying agent architecture.

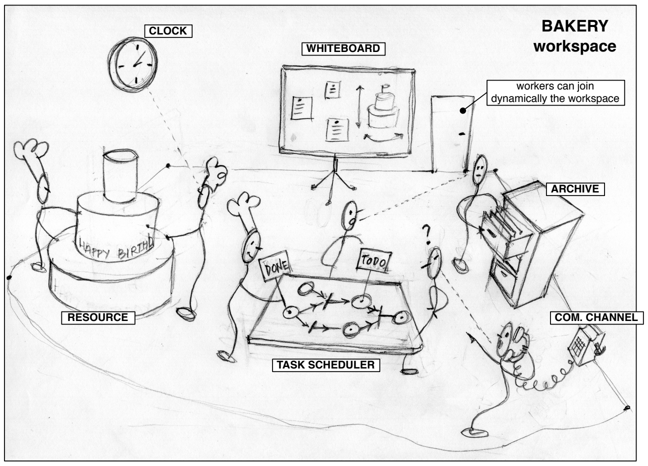


Figure 1. Abstract representation of the A&A metaphor in the context of a bakery.

III. Agent-Oriented Abstractions for Computer Programming

Quoting Lieberman [18], *"The history of Object-Oriented Programming can be interpreted as a continuing quest to capture the notion of abstraction – to create computational artifacts that represent the essential nature of a situation, and to ignore irrelevant details"*. In that perspective, in this section we identify and discuss a core set of concepts and abstractions introduced by agent-oriented programming. While most of these concepts already appeared in literature in different contexts, our aim here is to highlight their value for framing a conceptual space and an abstraction layer useful for defining general-purpose programming languages.

A. The Background Metaphor

Metaphors play a key role in computer science, as means for constructing new concepts and terminology [31]. In the case of objects in OOP, the metaphor is about real-world objects. Like physical objects, objects in OOP can have properties and states, and like social objects, they can communicate as well as respond to communications. In the case of actors [6], similarly, the inspiration is clearly more anthropomorphic, and a variety of anthropomorphic metaphors influenced its development [29][17].

The inspiration for the agent-oriented abstraction layer that we discuss in this paper is anthropomorphic too and refers to the A&A (Agents and Artifacts) conceptual model [20], which takes human organizations as main reference. Figure 1 shows an example of such metaphor, represented by a human working environment, a *bakery* in particular. It is a system where articulated concurrent and coordinated activities take place, distributed in time and space, by people working inside a common environment. Activities are explicitly targeted to some objectives. The complexity

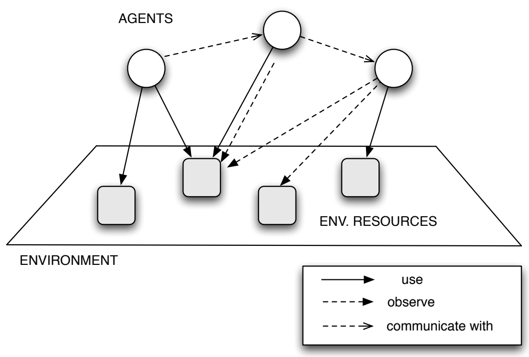


Figure 2. Abstract representation of an agent-oriented program composed by agents working within an environment.

of work calls for some division of labor, so each person is responsible for the fulfillment of one or multiple tasks. Interaction is a main dimension, due to the dependencies among the activities. Cooperation occurs by means of both direct verbal communication and through tools available in the environment (e.g., a blackboard, a clock, the task scheduler). So the environment – as the set of tools and resources used by people to work – plays a key role in performing tasks efficiently. Besides tools, the environment hosts resources that represent the co-constructed results of people work (e.g., the cake).

Following this metaphor, we see a program – or software system – as a collection of autonomous agents working and cooperating in a shared environment Figure 2: on the one side, agents (like humans) are used to represent and modularize those parts of the system that need some level of autonomy and pro-activity—i.e., those parts in charge to autonomously accomplish the tasks in which the overall labor is split; on the other side, the environment is used to represent and modularize the non-autonomous functionalities that can be dynamically composed, adapted and used (by the agents) to perform the tasks.

A main feature of this approach is that it promotes a *decentralized mindset* in programming, as also considered by Resnick in [24]. Such a mindset has two main cornerstones.

The first one is the *decentralization and encapsulation of control*: there is not a unique locus of control in the system, which is instead decentralized into agents. It is worth remarking that here we are assuming a logical point of view over decentralization—not strictly related to, for instance, physical threads or processes. The agent abstraction extends the basic encapsulation of state and behavior featured by objects by including also encapsulation of control, which is fundamental for defining and realising agent *autonomous* behaviour.

The second cornerstone is the interaction dimension which includes coordination and cooperation. There are two basic orthogonal ways of interacting: direct communication among agents based on high-level asynchronous message passing and environment-mediated interaction (discussed in Subsection III-D) exploiting the functionalities provided by environment resources.

B. Structuring Active Behaviors: Tasks and Plans

Decentralization and encapsulation of control, as well as direct communication based on message passing, are main properties also of actors, as defined in [6]. The actor model, however, does not provide further concepts useful to *structure* the autonomous behavior, besides a simple notion of *behavior*. This is an issue as soon as we consider the development of large or simply not naive active entities. To this end, the agent abstraction extends the actor one introducing further high-level notions that can be effectively exploited to organize agent autonomous behavior, namely *tasks* and *plans*.

The notion of task is introduced to specify a unit of work that has to be executed—the objective of agents' activities. So, an agent acts in order to perform a task, which can be possibly assigned dynamically. The same agent can be able to accomplish one or more types of task, and the *type* of the agent can be strictly related to the set of task types that it is able to perform.

Conceptually, an agent is hence a computing machine that, given the description of a task to execute, it repeatedly chooses and executes *actions* so as to accomplish that task. If the task concept is used as a way to define *what* has to be executed, the set of actions to be chosen and performed represents *how* to execute such tasks. The first-class concept used to represent one such set is the *plan*. So the agent programmer defines the behavior of an agent by writing down the plans that the agent can dynamically combine and exploit to perform tasks. For the same task, there could be multiple plans, related to different contextual conditions that can occur at runtime.

On the one side, tasks and plans can be used to define the contract explicitly stating what jobs the agent is able to do; on the other side, they are used (by the agent programmer) to structure and modularize the description of how the agent is able to do such jobs, organizing plans in sub-plans.

This approach makes it possible to frame a smooth path in defining different levels of abstraction in specifying plans and, correspondingly, different levels of autonomy of agents. At the base level, a plan can be a detailed description of the sequence of actions to execute. In this case, task execution is fully pre-defined, since the programmer is charged with the entire task specification; the level of autonomy of the agent is limited in selecting the plan among the possible ones specified by the programmer. In a slightly more complex case, a plan could be the description of a set of possible actions to perform, and the agent uses some criteria at runtime to select which one to execute. This enhances the level of autonomy of the agent with respect to what strictly specified by the programmer. An even stronger step towards

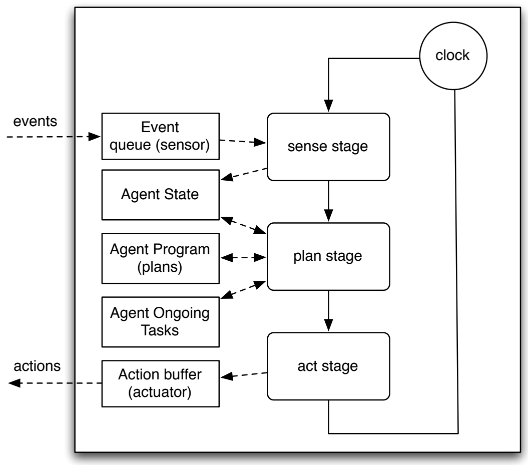


Figure 3. Conceptual representation of an agent architecture, with in evidence the stages of the execution cycle.

autonomy is given by the case in which a plan is just a partial description of the possible actions to execute, and the agent dynamically infers the missing ones by exploiting information about the ongoing tasks, and about the current knowledge of its state and the state of the environment.

C. Integrating Active and Reactive Behaviours: The Agent Execution Cycle

More and more the development of applications calls for flexibly integrating active and reactive computational behaviors, an issue which is strongly related to the problem of integrating thread-based and event-based architectures [14]. Active behaviors are typically mapped on OS threads, and the asynchronous suspension/stopping/control of thread execution in reaction to an event is an issue in high-level languages. So, for instance, in order to make a thread of control aware of the occurrence of some event – to be suspended or stopped – it is typically necessary to “pollute” its block of statements with multiple tests spread around.

In the case of agents, this aspect is tackled quite effectively by the control architecture that governs their execution, which can be considered both *event-driven* and *task-driven*. The execution is defined by a control loop composed by a possibly non-terminating sequence of execution cycles. Conceptually, an execution cycle is composed by three different stages (see Figure 3):

- *sense* stage – in this stage the internal state of the agent is updated with the *events* collected in the agent event queue. So this is the stage in which inputs generated by the environment during the previous execution cycle are fetched.
- *plan* stage – in this stage the next action to execute is chosen, based on the current state of the agent, the agent plans and agent ongoing tasks; additionally, agent state is also updated to reflect such a choice.
- *act* stage – in this stage the actions selected in the *plan* stage are executed.

The agent machine continuously executes these three stages, performing one execution cycle at each logical clock tick. Conceptually, the agent control flow is never blocked—actually it can be in idle state if, for instance, the executed plan states that no action has to be executed until a specific event is fetched in the sense stage. This architecture easily allows, for instance, for suspending a plan in execution and execute another plan to handle an event suddenly detected in the sense stage.

While in principle this makes an agent machine less efficient than machines without such loops, this architecture allows to have a specific point to balance efficiency and reactivity thanks to the opportunity to define proper atomic actions. Besides, in practice, by carefully design the execution cycle architecture, it is possible to minimize the overheads – for instance by avoiding to cycle and consuming CPU time if there aren’t actions to be executed or new events to be processed – and eventually completely avoid overheads when needed —for instance, by defining the notion of atomic (not interruptible) plan, whose execution would be as fast as normal procedures or methods in traditional imperative languages.

D. “Something is Not an Agent”: the Role of the Environment Abstraction

Often programming paradigms strive to provide a single abstraction to model every component of a system. This happens, for instance, in the case of actor-based approaches. In Erlang [7] for instance, which is actor-based, every macro-component of a concurrent system is a process, which is the actor counterpart. This has the merit of providing uniformity and simplicity, indeed. At the same time, the perspective in which everything is an active, autonomous entity is not always effective, at least from an abstraction point of view. For instance, it is not really natural to model as active entities either a shared bounded-buffer in producers/consumers architectures or a simple shared counter in a concurrent programs. In traditional thread-based systems such entities are designed as monitors, which are passive.

Switching to an agent abstraction layer, there is an apparent uniformity break due to the notion of *environment*, which is a first-class concept defining the context of agent tasks, shared among multiple agents.

From a designer and programmer point of view, the environment can be suitably framed as such non-autonomous part of the system which be used to encapsulate and modularize those functionalities and services that are eventually shared and exploited by the autonomous agents at runtime. More specifically, by recalling the human metaphor, the environment can be framed as the set of *resources* and *tools* that are possibly shared and *used* by agents to execute their tasks. In that perspective, a bounded-buffer, a shared database etc. can be naturally designed and programmed as a shared resource populating the environment where – for instance – producers/consumers agents work.

E. Using and Observing the Environment

To be usable by agents, an environment resource provides a set of *operations* – that constitutes its *usage interface* – encapsulating some piece of functionality. Such operations are the basic actions that an agent can execute on instances of that resource type. So the set of actions that an agent can execute inside an environment depends on the set of resources that are available in that environment. Since resources can be created and disposed at runtime by agents, the agent action repertoire can change dynamically.

The execution of an operation (action) performed by an agent on a resource may complete with a success or a failure—so an explicit success/failure semantics is defined. Actions (operations) are performed by agents in the act stage of the execution cycle seen previously. Then, the completion of an action occurs asynchronously, and is perceived by the agent as a basic type of event, fetched in the sense stage. This can occur in the next execution cycle or in a future execution cycle, since the execution of an operation can be long-term. So, an important remark here is that the execution cycle of an agent *never blocks*, even in the case of executing actions that – to be completed – need the execution of further actions of other agents. This means that an agent, even if "waiting" for the completion of an action, can react to events perceived from the environment and execute a proper action, following what is specified in the plan.

Finally, aside to actions, *observable properties* and *observable events* represent the other side of agent-environment interaction, that is the way in which an agent gets input information from the environment. In particular, observable properties represent the observable state that an environment resource may expose, as part of its functionalities. The value of an observable property can be changed by the execution of operations of the same resource. A simple example is a counter, providing an inc operation (action) and an observable state given by an observable property called count, holding the current count value. By observing a resource, an agent automatically receives the updated value of its observable properties as percepts at each execution cycle, in the sense stage. Observable events represent possible signals generated by operation execution, used for making observable an information not regarding the resource state, but regarding a dynamic condition of the resource. Taking as a metaphor a coffee machine as environment resource, the display is an observable property, the beep emitted when the coffee is ready is an observable event. Choosing what to model as a property or as an event is a matter of environment design.

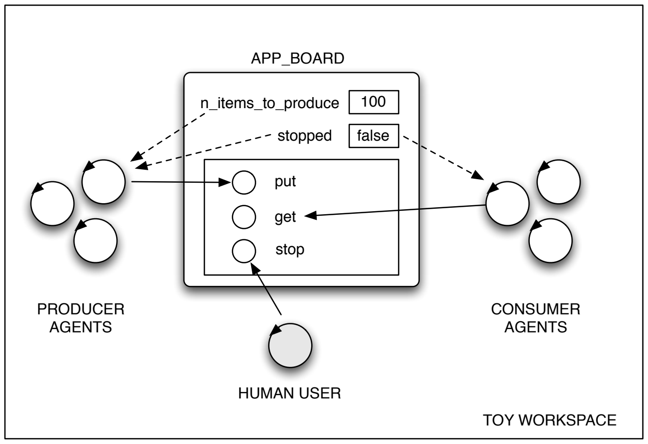


Figure 4. A toy workspace, with producer and consumer agents interacting by means of an app_board artifact.

IV. Evaluating the Idea with Existing Agent Technologies: The JaCa Platform

The aim of this section is to show more in practice some of the concepts described in the previous section. To this end, we will use existing agent technologies, in particular a platform called JaCa, which actually integrates two independent technologies: the Jason agent programming language [10] – for programming agents – and the CArtAgO framework [25], for programming the environment.

A. JaCa Overview

Following the basic idea discussed in Section III - a JaCa program is conceived as a dynamic set of autonomous agents working inside a shared environment, that they use, observe, adapt according to their tasks. The environment is composed by a dynamic set of environment resources which in CArtAgO are called "artifacts"—the term was inspired by Activity Theory and Distributed Cognition, where it is used to refer to any object that has been specifically designed to provide some functionality and which is used by humans to achieve their objective. Agents – by means of proper actions – can dynamically create and dispose artifacts, beside using them.

In the following, we introduce only those basic elements of agent and environment programming which are necessary to show the features discussed at the conceptual level in the previous section. To this end, we use a toy example which is about the implementation of a producers-consumers architecture, where a set of producer agents continuously and concurrently produce data items which must be consumed by consumer agents (see Figure 4). Further requirements – which make the example more interesting for our purposes – are that *(i)* the number of items to be produced is fixed, but the time for producing each item (by the different producers) is not known a priori; *(ii)* the overall process can be interrupted by the user anytime.

The task of producing items is divided upon multiple producer agents, acting concurrently—the same holds for consumer agents. To interact and coordinate the work, agents share and use an environment resource, the app_board artifact, which functions both as a buffer to collect items inserted by producers and to be removed by consumers and as a tool to control the overall process by the human user. The resource provides on the one side operations (actions for the agent) to insert (`put`), remove (`get`) items and to stop the overall activities (`stop`); on the other side, observable properties `n_items_to_produce` and `stopped`, keeping track of, respectively, the number of items still to be produced (which starts from an initial value and is decremented by the resource each time a new item is inserted) and the stop flag (initially false and set to true when the `stop` operation is executed).

In the following, first we give some glances about agent programming in Jason by discussing the implementation of a producer agent (see Table I), which must exhibit a pro-active behavior – performing cooperatively the production of items, up to the specified number – but also a reactive behavior: if the user stops the process, the agents must interrupt their activities. Then we briefly consider the implementation of the app_board artifact, to show in practice some elements of environment programming.

B. *Programming Agents in* Jason

Being inspired by the BDI (Beliefs-Desires-Intentions) architecture [23], the Jason language constructs that programmers can use can be separated into three main categories: *beliefs*, *goals* and *plans*. An agent program is defined by an initial set of *beliefs*, representing the agent's initial knowledge about the world, a set of *goals*, which corresponds to tasks as defined in Section III, and a set of *plans* that the agent can dynamically compose, instantiate and execute to achieve such goals.

In JaCa the beliefs of an agent can represent two types of knowledge:

- the agent internal state – an example is given by the `n_items_produced(N)` belief, which is used by a producer agent to keep track of the number of items produced so far;
- the observable state of the resources of the environment which the agent is observing – in the example, every producer agent observes the app_board artifact, which has two observable properties: `n_items_to_produce`, representing the number of items still to be produced, and `stopped`, a flag which is set if/when the process needs to be stopped.

An agent program may explicitly define the agent's initial belief-base and the initial task or set of tasks that the agent has to perform, as soon as it is created. In Jason tasks are called *goal* and are represented by Prolog atomic formulae prefixed by an exclamation mark. Referring to the example, the producer agent has an initial task to do, which is represented by the `!produce` goal. Actually, tasks can be assigned also at runtime, by sending to an agent an achieve-goal messages.

```
00  n_items_produced(0).
01  !produce.
02
03  +!produce
04    <- !setup;
05       !produce_items.
06
07  +!setup
08    <- focus("app_board").
09
10  +!produce_items : not n_items_to_produce(0)
11    <- !produce_item(Item);
12       put(Item);
13       -n_items_produced(N);
14       +n_items_produced(N+1);
15       !produce_items.
16
17  +!produce_items : n_items_to_produce(0)
18    <- !finalize.
19
20  +!produce_item(Item) <- ...
21
22  +!finalize : n_items_produced(N)
23    <- println("completed - items produced: ",N).
24
25  -!produce_items
26    <- !finalize.
27
28  +!stopped(true)
29    <- .drop_all_intentions;
30       !finalize.
```

Table I
A PRODUCER AGENT IN JASON.

Then, the main body of an agent program is given by the set of plans, which defines the pro-active and reactive behavior of the agent. The actions contained in a plan body can be split in two categories:

- *internal* actions, that are actions affecting only the internal state of the agent. Examples are actions to create sub-tasks (sub-goals) to be achieved (`!g`), to manage task execution – for instance, to suspend or abort the execution of a task – to update agent inner state – such as adding a new belief (`+b`), removing beliefs (`-b`);
- *external* actions, that are actions provided by the environment, to interact with artifacts. External actions include also communicative actions, which make it possible to communicate with other agents by means of message passing based on speech acts.

Referring to the example, the producer agent has a main plan (line 03-05), which is triggered by an event `+!produce` representing a new goal `!produce` to achieve. Since the agent has an initial `!produce` goal, then this plan will be triggered as soon as the agent is booted. By means of an internal action `!g`, the main plan generates two further subgoals to be achieved sequentially: `!setup` and `!produce_items`.

The plan to handle `!setup` goal (line 07-08) exploits a predefined action called focus to start observing the app_board artifact. Then, two plans are specified for handling the goal !produce_items. One (line 10-15) is executed if there are still items to produce—i.e., if the agent has not the belief n_items_to_produce(0). Note that the value of this belief depends on the current state of the app_board resource. This plan first produces a new item (subtask !produce_item), then inserts the item in the buffer by means of a put action, whose effect is to execute the put operation on the resource; if this action succeeds, the plan goes on by updating the belief n_items_produced incrementing the number of items produced and generates a new subgoal !produce_items to repeat the task. Actually, when executing an external action – such as put – it is possible to explicitly denote the artifact providing that action, in order to avoid ambiguities, by means of Jason annotations: put(Item) [artifact_name("app_board")];.

The other plan (line 17-18) is executed if there are no more items to produce: in this case the !finalize task is executed, which prints on the console the number of items produced by the agent.

The reactive behavior of an agent can be realized by plans triggered by a belief addition/change/removal – corresponding to changes in the state of the environment – and by the failure of a plan in achieving some goal. In the example, the producer agent has a plan (line 28-30) which is executed when the belief stopped about the observable property of the artifact is updated to true. This means that the user wants to interrupt and stop the production. So the plan stops and drops all the other possible plans in execution – using an internal action .drop_all_intention – and the !finalize subtask is executed.

Finally, the producer agent has also a plan (line 25-26) to react to the failure of the !produce_items task, which is expressed by the event -!produce_items. This can happen when the agent, believing that there are still items to be produced, starts the plan to produce a new item and tries to insert it in the buffer. However, the put action fails because other agents produced in the meanwhile the missing items.

The semantics of the execution of plans reacting to events is defined by Jason reasoning cycle [10], which is a more articulated version of the execution cycle described in Section III. In particular, the plan stage in this case includes multiple steps, to select – given an event – a plan to be executed. So an agent can have multiple plans in execution but only one action at a time is selected (in the plan stage) and executed (in the act stage). A detailed description of the cycle – as well as of the Jason syntax – can be found in [10].

```
public class AppBoard extends Artifact {

  private LinkedList<Object> items;
  private int bufSize;

  void init(int bufSize, int nItemsToProd){
    items = new LinkedList<Object>();
    this.bufSize = bufSize;
    defineObsProperty("n_item_to_produce",nItemsToProd);
    defineObsProperty("stopped",false);
  }

  @OPERATION void put(Object obj){
    await("bufferNotFull");
    ArtifactObsProperty stopped =
                            getObsProperty("stopped");
    if (!stopped.booleanValue()){
      items.add(obj);
      ArtifactObsProperty p  =
           getObsProperty("n_item_to_produce");
      p.updateValue(p.intValue() - 1);
    } else {
     failed("no_more_items_to_produce");
    }
  }

  @GUARD boolean bufferNotFull(){
    return items.size() < nmax;
  }

 @OPERATION void get(OpFeedbackParam<Object> result){
    await("itemAvailable");
    Object item = items.removeFirst();
    result.set(item);
  }

 @GUARD boolean itemAvailable(){
    return items.size() > 0;
  }

 @OPERATION void stop(){
    updateObsProperty("stopped",true);
  }
}
```

Table II
THE IMPLEMENTATION OF THE APP_BOARD IN CARTAGO.

C. Programming the Environment in CArtAgO

The implementation of the app_board artifact is shown in Table II. Being CArtAgO a framework on top of the Java platform, artifact-based environments can be implemented using a Java-based API, exploiting the annotation framework. Here we don't go too deeply into the details of such API, we just introduce the main concepts that have been mentioned in Section III; for more information, the interested reader can refer to CArtAgO papers [25] and the documents that are part of CArtAgO distribution [2].

In CArtAgO, an artifact type can be defined by extending a base Artifact class. Artifacts are characterized by a usage interface containing a set of operations that agents can execute to get some functionalities. In the example, the artifact app_board provides three operations: put, get and stop. The put operation inserts a new element in the buffer – decrementing the number of items to be produced – if the stopped flag has not been set, otherwise the operation (action) fails. The get operation removes an item from the buffer, returning it as a feedback of the action. The stop operation sets the stopped observable property to true.

Operations are implemented by methods annotated with @OPERATION. The init method is used as constructor of the artifact, getting the initial parameters and setting up the initial artifact state. Inside an operation, guards can be specified (await primitive), which suspend the execution of the operation until the specified condition over the artifact state (represented by a boolean method annotated with @GUARD) holds. In the example, the put operation can be completed only when the buffer is not full (bufferNotFull guard) and the get one when the buffer is not empty (bufferNotEmpty guard). The execution of operations inside an artifact is transactional: among the other things, this implies that at runtime multiple operations can be invoked concurrently on an artifact but only one operation can be in execution at a time–the other ones are suspended. On the agent side, when executing an external action, the agent plan is suspended until the corresponding artifact operation has completed (i.e., the action completed). Then, the action succeeds or fails when (if) the corresponding operation has completed with success or failure. It is worth noting that, in the meanwhile, the agent execution cycle can go on, making it possible for the agent to get percepts and select and perform other actions.

Besides operations, artifacts typically have also a set of observable properties (n_items_to_produce and stopped in the example), as data items that can be perceived by agents as environment state variables. Instance fields of the class – instead – are used to implement the non observable state of the artifact—for instance, the list of items items in the example. Observable properties can be defined, typically during artifact initialization, by means of the defineObsProperty primitive, specifying the property name and initial value (line 08-09). Inside operations, observable properties value can be inspected and changed dynamically by means of two basic primitives: getObsProperty to retrieve the current value of an observable property (see, for instance, line 14 and 18) and updateObsProperty to update the value (line 19).

Besides observable properties, an artifact can make it observable also events occurring when executing operations. This can be done by using a signal primitive, specifying the type of the event and a list of actual parameters. For instance, signal("my_event", "test",0) generates an observable event my_event("test",0). In the app_board example, to notify the stop we could generate a stopped signal in the stop operation, instead of using an observable property. Observable events are perceived by all agents observing the artifact—which could react to them as in the case of observable property change.

D. Using JaCa In Real-World Application Contexts

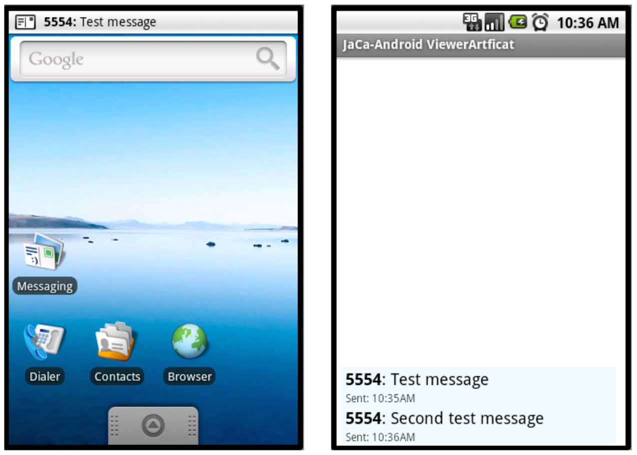
In order to stress the benefits but also the weaknesses of the approach, we are applying this programming model and technology in different application domains.

One is the development of distributed applications based on Service-Oriented Architectures and Web Services in particular. In that context, agents and multi-agent systems are deserving increasing attention both from the applicative viewpoint, as an effective technique to build complex services and applications dynamically composing and orchestrating services [19], and from the foundational viewpoint, as a reference meta-model for the service-based approach, as suggested by the W3C document about Web Services Architecture [3]. To this end, programming models and platforms are needed that make it possible to build SOA/WS applications as agent-oriented systems in a systematic way, exploiting the existing agent languages and platforms to their best, while enabling their co-existence and fruitful cooperation. In that context, we devised a library of artifacts on top of the JaCa platform, enabling the development of SOA/WS applications in terms of workspaces populated by agents and artifacts. Agents encapsulate the responsibility of the execution and control of the business activities and tasks that characterize the SOA-specific scenario, while artifacts encapsulate the business resources and tools needed by agents to operate in the application domain. In particular, artifacts in this case are exploited to model and engineer those parts in the agent world that encapsulate Web Services aspects and functionalities – eventually wrapping existing non-agent-oriented code – to be used, but also changed and adapted by agents at runtime, by need. First results of this work are available here [21].

We are also investigating the approach for the engineering of advanced mobile computing applications, in particular for pervasive and context-aware computing scenarios. To this end, JaCa has been ported on the Android platform [4], enabling the development of Android applications using agent-oriented programming. The project is called JaCa-Android [5]. Actually, besides porting the technology, JaCa-Android includes a library of artifacts that allows agents running into an Android application to seamlessly access and exploit all the features provided by the smartphone and by the Android SDK. Just to have a taste of the approach, Table III shows a snippet of an agent playing the role of smart user assistant, with the task of managing the notifications related to the reception of SMS messages: as soon as an SMS is received, a notification must be shown to the user. A SMSArtifact artifact is used to manage SMS messages, in particular this artifact generates an observable event sms_received each time a new SMS is received. A ViewerArtifact artifact is used to show SMS messages on the screen and to keep track – by means of the state observable property – of the current status of the viewer, that is if it is currently visualized by the user on the smartphone screen or not. Finally, a StatusBarArtifact artifact is used instead to show messages on the Android status bar, providing a showNotification operation to this end. Depending on

```
00  !init.
01
02  +!init
03  <-  focus("SMSArtifact");
04      focus("SMSArtifact");
05      focus("ViewerArtifact").
05
07  +sms_received(Source, Message)
08    : not (state("running") & session(Source)) <-
09    showNotification("jaca.android:drawable/notification",
10       Source, Message, "jaca.android.sms.SmsViewer", Id);
11    +session(Source, Id).
12
13  +sms_received(Source, Message)
14    : state("running") & session(Source)
15    <-  append(Source, Message).
```

Table III
SOURCE CODE OF THE JASON AGENT THAT MANAGES THE SMS NOTIFICATIONS.



5554: Test message
Sent: 10:35AM
5554: Second test message
Sent: 10:36AM

Figure 5. The two different kinds of SMS notifications: (*a*) notification performed using the standard Android status bar, and (*b*) notification performed using the `ViewerArtifact`.

what the user is actually doing and visualizing, the agent shows the notification in different ways. The behavior of the agent, once completed the initialization phase (lines 00-05), is governed by two reactive plans. The first one (lines 7-11) is applicable when a new message arrives and the `ViewerArtifact` is not currently visualized on the smartphone's screen. In this case, the agent performs a `showNotification` action to notify the user of the arrival of a new message using the status bar (Figure 5, *(a)*). The second plan instead (lines 13-15) is applicable when the `ViewerArtifact` is currently displayed on screen and therefore the agent could notify the SMS arrival by simply appending the SMS to the received message list showed by the viewer (Figure 5, *(b)*): this is done by executing the `append` operation provided by `ViewerArtifact`.

From the example, it should be clear that for a developer able to program using the JaCa programming model, moving from one application context to another is a quite straightforward experience. Indeed, she can continue to engineer the business logic of the applications by suitably defining the Jason agent's behavior, and it only need to acquire the ability to work with the artifacts that are specific of the new application context.

E. Current Limitations

On the one side, JaCa allows to exploit in practice some of the benefits of agent-orientation for computer programming described in Section III; on the other side, it suffers of some limitations that we aim at tackling in our future work. Here we consider three main ones.

First, Jason lacks a strong notion of *type*, both for defining the abstract data types used in the programs and for typing agents themselves. This makes agent programs error-prone – some errors are caught only at runtime – and features like inheritance, sub-classing, polymorphism cannot be exploited when developing agents. This is a quite strong limitation due to the fact that such features are the key for providing reusability of the code produced by the developers and therefore are quite essential for: (*i*) the engineering of real-world applications and (*ii*) for the diffusion of the AOP as a mainstream paradigm.

Then, a more seamless integration between the model/platform with the Object-Oriented and Functional programming layer is needed. Currently, for using objects/functions or for integrating any kind of software library (e.g., a library for XML-manipulation), we need to use some sort of wrap mechanism for making them available when programming agents. Now we can realize this sort of wrapping in two ways: (*i*) extending the set of Jason internal actions for directly provide to the agents the required features or (*ii*) encapsulating the required object-oriented/functional-oriented code inside proper artifacts operations.

Finally, Jason plan construct provides a quite weak support for modularizing agent programs. Currently the overall behavior of an agent is defined by a flat list of plans. The absence of a hierarchical structure for plans, explicitly relating plans with sub-plans, could make the understanding of complex agent behavior quite problematic.

V. CONCLUSION

In this paper, we discussed agent-oriented programming as an evolution of Object-Oriented Programming representing the essential nature of decentralized systems where tasks are in charge of autonomous computational entities, which interact and cooperate within a shared environment. We showed in practice some of the main concepts underlying the approach by exploiting the JaCa platform, which is based on existing agent-oriented technologies—the Jason language to program agents and CArtAgO framework to program the environment. However, we believe that, in order to stress and investigate the full value of the agent-oriented approach, a new generation of agent-oriented programming languages is needed, tackling main aspects that have not been considered so far in existing agent technologies –

being not related to AI but to the principles of software development. This is the core of our current and future work, in which we aim at both improving JaCa and eventually exploring the definition of new full-fledged agent-oriented programming languages – so independent from existing technologies – specifically designed since their conception for agent-oriented computing.

References

[1] The Foundation of Intelligent Physical Agents organization (FIPA) – http://www.fipa.org, last retrieved: July 5th 2011.

[2] CArtAgO project web site – http://cartago.sourceforge.net, last retrieved: July 5th 2011.

[3] W3C Web Service Architecture – http://www.w3.org/TR/ws-arch/, last retrieved: July 5th 2011.

[4] Android Platform web site – http://developer.android.com, last retrieved: July 5th 2011.

[5] JaCa-Android project web site – http://jaca-android.sourceforge.net/, last retrieved: July 5th 2011.

[6] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[7] J. Armstrong. Erlang. *Commun. ACM*, 53:68–75, September 2010.

[8] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[9] R. Bordini and J. Hübner. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *CLIMA VI*, volume 3900 of *LNAI*, pages 143–164. Springer, Mar. 2006.

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

[11] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Special Issue: Multi-Agent Programming*, volume 23 (2). Springer Verlag, 2011.

[12] R. H. Bordini, M. Dastani, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming Languages, Platforms and Applications - Volume 1*, volume 15. Springer, 2005.

[13] R. H. Bordini, M. Dastani, A. El Fallah Seghrouchni, and J. Dix, editors. *Multi-Agent Programming Languages, Platforms and Applications - Volume 2*. Springer, 2009.

[14] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

[15] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents™ — summary of an agent infrastructure. In *Proceedings of Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, held with the Fifth International Conference on Autonomous Agents (Agents 2001)*, 2001.

[16] N. R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.

[17] W. A. Kornfeld and C. Hewitt. The scientific community metaphor, 1981. MIT Artificial Intelligence Laboratory.

[18] H. Lieberman. The continuing quest for abstraction. In *ECOOP 2006*, volume 4067/2006, pages 192–197. Springer, 2006.

[19] M. N. Huhns, M. P. Singh, and M. e. a. Burstein. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):69–70, Nov. 2005.

[20] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17 (3), Dec. 2008.

[21] M. Piunti, A. Santi, and A. Ricci. Programming SOA/WS systems with BDI agents and artifact-based environments. In *MALLOW-AWESOME*, 2009.

[22] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In R. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming*. Kluwer, 2005.

[23] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *First International Conference on Multi Agent Systems (ICMAS95)*, 1995.

[24] M. Resnick. *Turtles, Termites and Traffic Jams. Explorations in Massively Parallel Microworlds*. MIT Press, 1994.

[25] A. Ricci, M. Piunti, and M. Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23:158–192, 2011.

[26] A. Ricci, M. Viroli, and G. Piancastelli. simpA: An agent-oriented approach for programming concurrent applications on top of java. *Science of Computer Programming*, 76(1):37 – 62, 2011.

[27] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach (second edition)*. Prentice Hall, 2010.

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

[29] B. C. Smith and C. Hewitt. A plasma primer, 1975. MIT Artificial Intelligence Laboratory.

[30] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue: Tomorrow's Computing Today*, 3(7):54–62, Sept. 2005.

[31] M. D. Travers. *Programming with Agents: New metaphors for thinking about computation*. Massachusetts Institute of Technology, 1996.

[32] M. Wooldridge. *An Introduction to Multi-Agent Systems*. John Wiley & Sons, Ltd, 2002.