A Programming Paradigm based on Agent-Oriented Abstractions

Alessandro Ricci
University of Bologna
Via Venezia 52, 47521 Cesena (FC), Italy
a.ricci@unibo.it

Andrea Santi
University of Bologna
Via Venezia 52, 47521 Cesena (FC), Italy
a.santi@unibo.it

***Abstract*—More and more the notion of *agent* appears in different contexts of computer science, often with different meanings. Main ones are Artificial Intelligence (AI) and Distributed AI, where agents are exploited as a technique to develop 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. The main contribution of the work is the definition of a conceptual space framing the basic features that characterize the agent-oriented approach as a programming paradigm, and its validation in practice by using a platform called JaCa, with real-world programming examples.**

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

I. Introduction

More and more the notion of *agent* appears in different contexts of computer science, often with different meanings. In the context of Artificial Intelligence (AI) or Distributed Artificial Intelligence (DAI), agents and multi-agent systems are typically exploited as a technique to tackle complex problems and develop intelligent software systems [1][2][3]. In this paper, we discuss a further perspective, which aims at exploiting the value of agents and multi-agent systems as a *programming paradigm*, providing high-levels concepts and mechanisms that are effective to tackle main challenges that characterize modern and future programming, concerning e.g. concurrency, distribution, autonomy, adaptivity.

Concurrency, in particular, due to the widespread diffusion of multi-core technologies, is more and more an important topic 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 [4], which means that nowadays concurrency is an issue that cannot be ignored or overlooked even in everyday programming, being it more and more a must-have feature for improving performance and responsiveness of programs. 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" [4]. We argue that agent-oriented programming – as framed in this paper – provides such level of abstraction, providing a rich set of concepts and mechanisms.

Actually, the idea of agent-oriented programming is not new in the context of AI/DAI: the first paper about AOP is dated 1993 [5], and since then many agent programming languages have been proposed in literature [6][7][8]. The objective of AOP as introduced in [5] 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 the practice of programming. This is the direction that we aim at exploring in our work and in this paper, which is a revised and extended version of a previous contribution [9].

The remainder of the paper is organized as follows. After presenting related work (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 [10] and CArtAgO [11]. After giving an overview of the main JaCa elements, in Section V we discuss in detail some selected features of JaCa programming, which are relevant for the development of software systems, and in Section VI we provide an overview of the application domains where JaCa has been effectively applied so far. Finally, in Section VII we discuss the main features that are currently missing in existing agent technologies, paving the way to the design and development of a new generation of agent-oriented programming languages. Concluding remarks are provided in Section VIII.

II. Related Work on Agent-Oriented Programming

As mentioned in the introduction, most of the agent-oriented programming languages and technologies – in particular those based on cognitive model/architectures such as the Belief-Desire-Intention (BDI) one [12] – have been introduced in the context of (Distributed) Artificial Intelligence [6][7][8]. Besides, 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) [13], a FIPA-compliant [14] 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 an explicit notion of shared programmable environments – perceived and acted upon by agents – based on the A&A (Agents and Artifacts) conceptual model [15], described in next sections.

Another example of Java-based agent-oriented framework is simpA [16], 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 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 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 [17], 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 [18] is a FIPA compliant framework based on Java and XML, but adopting the BDI as underlying agent architecture.

III. An Agent-Oriented Abstraction Layer

Quoting Lieberman [19], *"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 programming complex software systems.

A. The Background Metaphor

Metaphors play a key role in computer science, as means for constructing new concepts and terminology [20]. 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.

The inspiration for the agent-oriented abstraction layer that we discuss in this paper is anthropomorphic and refers to the A&A (Agents and Artifacts) conceptual model [15], which takes human organizations as main reference. Fig. 1 (on the left) 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 of the 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). Activity Theory [21] and distributed cognition [22] remark the fundamental role that such *artifacts* (i.e., resources and tools) have in human work and organization, both as media to enable and make it efficient communication, interaction and coordination and, more generally, to extend human cognitive and practical capabilities [23].

Following this metaphor, we see a program – or software system – as a collection of autonomous agents working and cooperating in a shared environment (Fig. 1): 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 – called *artifacts* in Activity Theory – 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 control mindset* in programming [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,

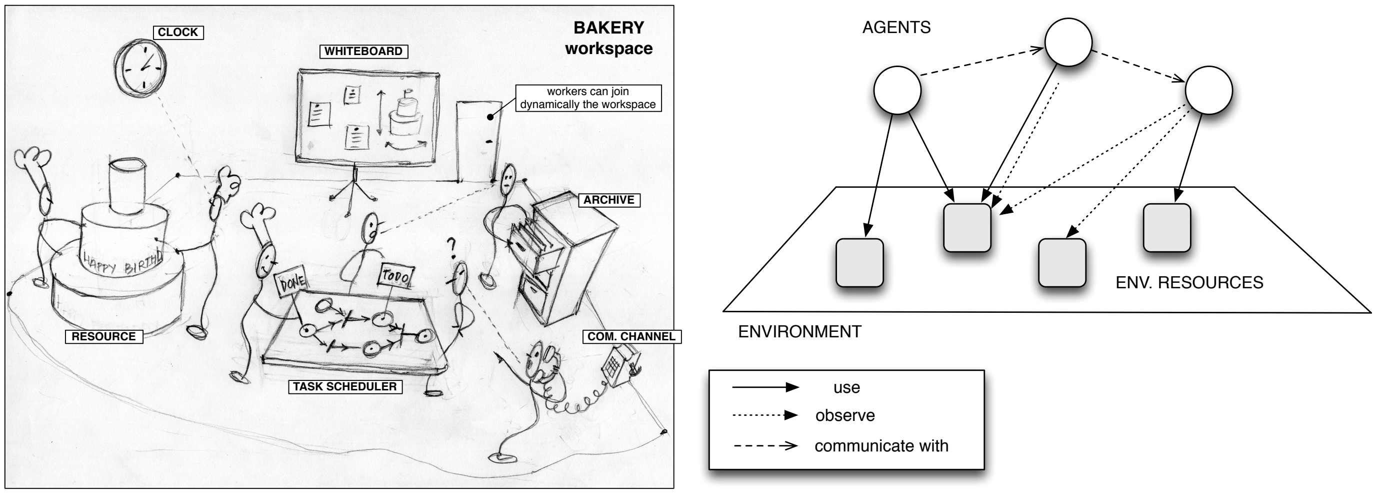


Fig. 1. *(Left)* Abstract representation of the A&A metaphor in the context of a bakery. *(Right)* Abstract representation of an agent-oriented program composed by agents working within an environment.

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 [25]. 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 – including those to react to relevant events – represent *how* to execute such task. 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 provides a complete specification of the plan; 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 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 strongly related to the problem of integrating thread-based and event-based architectures [26]. 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 Fig. 2):

- *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 – including messages sent by the other agents – 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 are no 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 [27] for example, 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 concurrent programs. In traditional thread-based systems such entities are designed as monitors, which are passive.

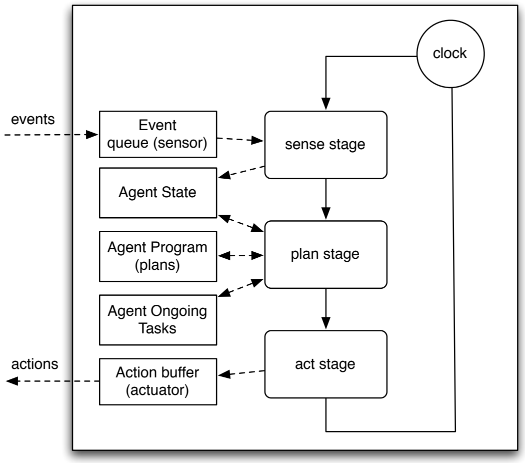


Fig. 2. Conceptual representation of an agent architecture, with in evidence the stages of the execution cycle.

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 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 objects functioning as *resources* and *tools* that are possibly shared and *used* by agents to execute their tasks. In order to avoid ambiguity with *objects* as defined in Object-Oriented Programming, here we will refer to these environment entities as *artifacts*, following our inspiring metaphor and adopting the terminology typically used in Activity Theory and Distributed Cognition. In that perspective, a bounded-buffer, a shared data-base etc. can be naturally designed and programmed as artifacts populating the environment where – for instance – producers/consumers agents work. Differently from agents, artifacts conceptually are not meant to be used to represent and implement autonomous / pro-active / re-active / task-oriented computational entities, but – more similar to passive objects or components or services – entities providing some functionality through a proper interface, that can be perceived and accessed by agents though actions.

E. Using and Observing the Environment

To be usable by agents, an artifact provides a set of *operations* – that constitute its *usage interface* – encapsulating some piece of functionality. Such operations are the basic actions that an agent can execute on instances of that artifact type. So the set of actions that an agent can execute inside an environment depend on the set of artifacts that are available in that environment. Since artifacts 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 an artifact 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.

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 artifact may expose, as part of its functionalities. The value of an observable property can be changed by the execution of operations of the same artifact. 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 an artifact, 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 artifact state, but regarding a dynamic condition of the artifact. Taking as a metaphor a coffee machine as an artifact, 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.

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 [11], for programming the environment.

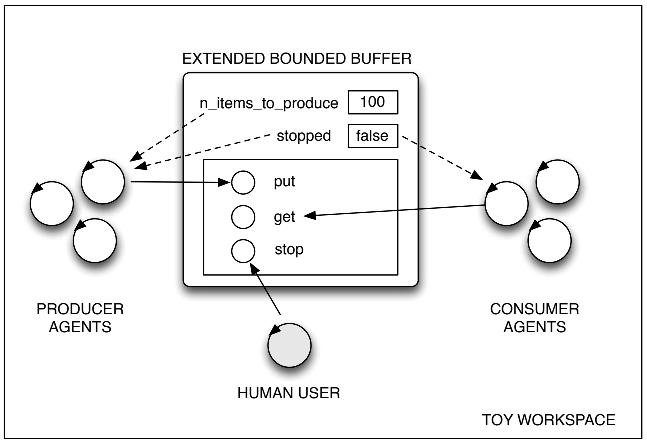


Fig. 3. A toy workspace, with producer and consumer agents interacting by means of a `ExtBBuffer` artifact.

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 artifacts, as computational entities that agents can dynamically create and dispose, beside using and observing them.

In the following, we introduce only those basic elements of agent and environment programming that are necessary to show the features discussed at the conceptual level in the previous section. To this end, we use a toy example about the implementation of a producers-consumers architecture, where a set of producer agents continuously and concurrently produce data items that must be consumed by consumer agents (see Fig. 3). Further requirements 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 `ExtBBuffer` 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 artifact 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, it provides 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 artifact 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 Fig. 4), 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. For completeness, also the source code of the consumer agent is reported (Fig. 5). Then we briefly consider the implementation of the `ExtBBuffer` artifact, to show in practice some elements of environment programming.

B. Programming Agents in Jason

Being inspired by the BDI (Beliefs-Desires-Intentions) architecture [12], 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 correspond 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. Logic programming is used to uniformly represent any piece of data and knowledge inside the agent program, beliefs and goals in particular.

Beliefs are represented as Prolog-like facts – that are atomic logical formulae – and represent the agent knowledge about:

- Its 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. Initially `N` is zero, and then it is dynamically updated by the agent in plans, by means of specific internal actions.
- The observable state of the artifacts that the agent is observing—in the example, every producer agent observes the `sharedBuffer` 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.

At design time the agent developer may want to define the agent's initial belief-base, by specifying some initial beliefs: then, beliefs can be added or removed at runtime, according to the agent changes to its state and to the resources that the agent dynamically decides to observe.

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 goals – i.e., tasks – 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 achieve-goal messages.

Then, the main body of an agent program is given by the set of plans, which define the pro-active and reactive behavior of the agent. Agent plans are described by rules of the type `Event : Context <- Body`, where `Event` represents the specific event triggering the plan, `Context` is a boolean expression on the belief base, indicating the conditions under which the plan can be executed once it has been triggered, and `Body` specifies the sequence of actions to perform, once the plan is executed. The actions contained in a plan body can be split in three 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`). Internal actions include also a set of primitives that allow for managing Java objects – which is the data model supported by CArtAgO – on the Jason side: so it is possible to create new objects (`cartago.new_obj`), invoke methods on objects (`cartago.invoke_obj`), etc.) and other related facilities (the prefix `cartago.` is used to identify in Jason the library to which the specific actions belong to).
- *External* actions, that are actions provided by the environment to interact with artifacts—as will be detailed in next section, these actions correspond to the operations provided by artifacts and included in artifact interfaces: so the repertoire of the actions of an agent is dynamic and depends on the number and type of artifacts available in the environment;
- *Communicative actions* (`.send`,`.broadcast`), 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 (lines 8-10), which is triggered by an event `+!produce` representing a new goal `!produce` to achieve. Since the agent has an initial `!produce` goal (line 4), 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 (lines 12-14) creates a new instance called `sharedBuffer` of type `ExtBBuffer` by means of a predefined action called `makeArtifact`, and then starts observing it by executing the predefined action `focus` specifying its identifier. This plan fails if the artifact had been already created (by another producer), generating a `-!setup` goal failure event: a plan managing the failure is specified (lines 16-18), which simply finds out the exact identifier of the existing artifact and starts observing it.

Then, two plans are specified for handling the goal `!produce_items`. One (lines 20-25) 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 `sharedBuffer` artifact. 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 artifact; 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

```
1   /* Producer agent */
2
3   n_items_produced(0).  /* initial belief */
4   !produce.             /* initial goal */
5
6   /* plans */
7
8   +!produce
9     <- !setup;
10       !produce_items.
11
12  +!setup
13    <- makeArtifact("sharedBuf","ExtBBuffer",[],Id);
14       focus(Id).
15
16  -!setup
17    <- lookupArtifact("sharedBuf",Id);
18       focus(Id).
19
20  +!produce_items : not n_items_to_produce(0)
21    <- !produce_item(Item);
22       put(Item);
23       -n_items_produced(N);
24       +n_items_produced(N+1);
25       !produce_items.
26
27  +!produce_items : n_items_to_produce(0)
28    <- !finalize.
29
30  +!produce_item(Item) <- ...
31
32  +!finalize : n_items_produced(N)
33    <- println("completed - items produced: ",N).
34
35  -!produce_items
36    <- !finalize.
37
38  +stopped(true)
39    <- .drop_all_intentions;
40       !finalize.
```

Fig. 4. Source code of a producer agent.

```
/* Consumer agent */

!consume.

+!consume: true
  <- ?bufferReady;
     !consumeItems.

+!consumeItems
  <- get(Item);
     !consumeItem(Item);
     !consumeItems.

+!consumeItem(Item) <- ...

+?bufferReady : true
  <- lookupArtifact("sharedBuffer",_).

-?bufferReady : true
  <-.wait(50);
    ?bufferReady.
```

Fig. 5. Source code of a consumer agent.

```
/* Main of the multi-agent program  */

MAS prodcons {
  environment: c4jason.CartagoEnvironment

  agents:
    producer agentArchClass c4jason.CAgentArch #10;
    consumer agentArchClass c4jason.CAgentArch #10;
}
```

Fig. 6. Main configuration file of the producers-consumers program.

ambiguities, by means of Jason annotations: `put(Item) [artifact_name("sharedBuffer")];`. The other plan (lines 27-28) is executed if there are no more items to produce—the `n_items_to_produce` belief referred in the plan context contains the updated value of the corresponding observable property in the artifact. In this case the `!finalize` task is executed, and it prints on standard output the number of items produced by the agent. The `println` action corresponds to the operation with the same name provided by an artifact called `console`, which is available by default in every workspace.

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 (lines 38-40) 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 (lines 35-36) 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] (shown in Fig. 7), 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). By executing an action, a plan is suspended until the action is completed (with success or failure). A detailed description of the cycle – as well as of the Jason syntax – can be found in [10].

C. Programming the Environment in CArtAgO

The implementation of the `ExtBBuffer` artifact is shown in Fig. 8. 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 [11] and the documents that are part of CArtAgO distribution [28].

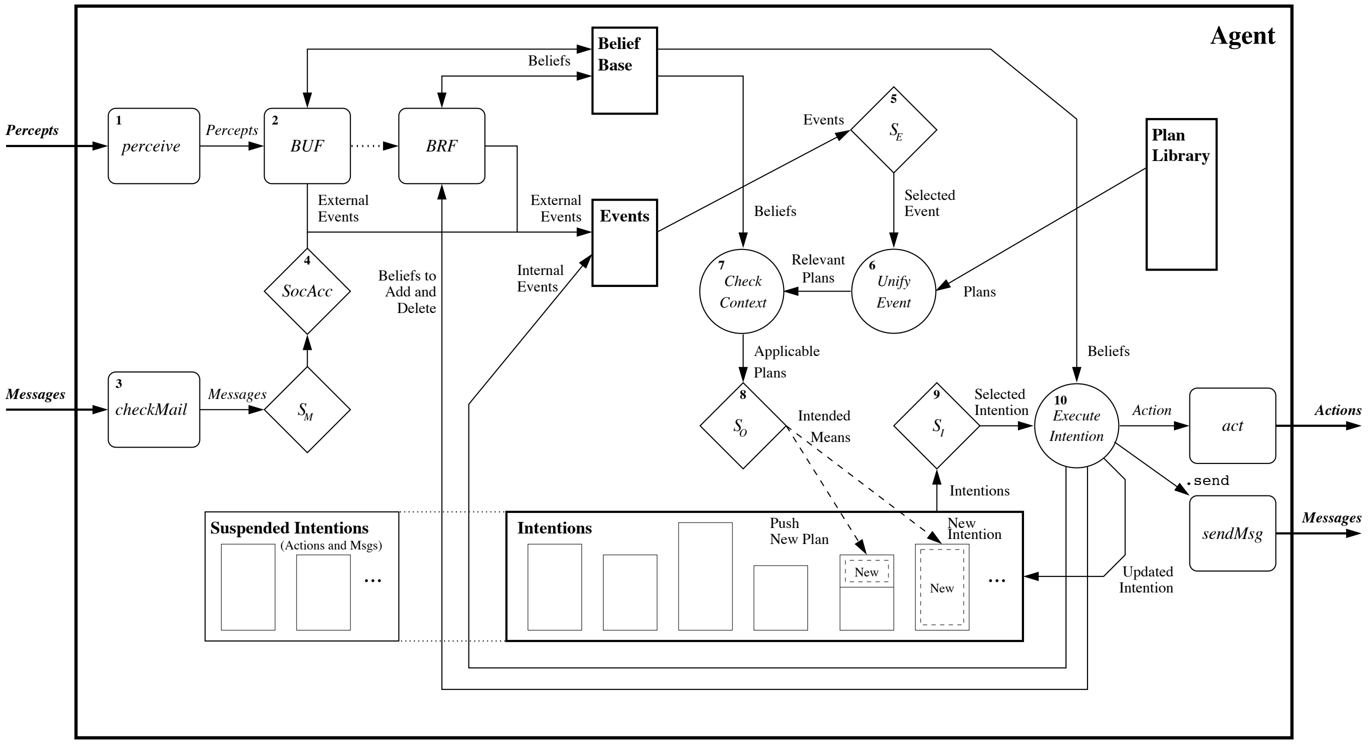


Fig. 7. A representation of Jason reasoning cycle (taken from [10]). In the first steps, the environment is perceived (step 1) and the beliefs about the state of the environment updated (step 2), by means of a customizable belief-update function (BUF). Besides input information from the environment, beliefs are updated also with messages possibly sent by other agents (step 3), filtered according some criteria defining "socially acceptable" messages (step 4). Then, updates to the belief base generate external events, appended in the event queue. This concludes the *sense* stage. Then, in the *plan* stage events are considered one by one (step 5), and for each one a relevant and applicable plan is selected (step 5 and 6), if available, from the plan library. If a plan is found, a new intention is instantiated (step 8), representing the plan in execution. The plan stage is completed by selecting the next action to do from one of the ongoing intentions (step 9). Finally, in the *act* stage the selected action is executed (step 10), and the cycle starts again.

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 `ExtBBuffer` 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.

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 (lines 11-12). Inside operations, observable properties value can be inspected and changed dynamically by means of primitives such as: `getObsProperty`, to retrieve the current value of an observable property (see, for instance, lines 18 and 22), `updateObsProperty` to update the value, or `updateValue` on an `ObsProperty` object, once the property has been retrieved with `getObsProperty` (line 23).

Besides observable properties, an artifact can make it ob-

```
import cartago.*;

public class ExtBBuffer 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);
  }
}
```

Fig. 8. Source code of the `ExtBBuffer` artifact.

servable 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 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.

Java objects and primitive data types are used as data model binding the agent and artifact layers, in particular to encode parameters in operations, fields in observable properties and signals.

To summarize, operations are computational processes occurring inside the artifact, possibly changing the observable properties and generating signals that are relevant for the agents using/observing the artifact. An operation is executed as soon as an agent triggers its execution – by executing the corresponding action. Given the transactional execution semantics adopted, only one operation can be in execution at a time—so no interferences and race conditions occur if multiple agents use concurrently the same artifact. Like in the case of monitors, other operations that are possibly concurrently triggered are blocked (suspended). The conditions that can be specified with the `await` command are conceptually similar to condition variables. Differently from the monitor case (with threads or processes), if an operation (action) is suspended, the agent that executed it is not: the execution cycle goes on, to eventually react to percepts and/or select and execute other actions from other plans.

Other features of the artifact model implemented in CArtAgO include: *(i)* the capability of *linking* together artifacts, making it possible for an artifact to execute operations (called linked operations) on other artifacts; *(ii)* the capability of triggering the execution of *internal* operations from other operations of the same artifact; and *(iii)* the capability of specifying for each artifact type a *manual*, i.e., a machine readable document containing the description of the functionalities provided by the artifacts of this type and the operating instructions, i.e., how to exploit such functionalities.

D. The Multi-Agent Program in the Overall

Finally, the *main* or entry point of a JaCa multi-agent program is given by a Jason source file – with extension `*.mas2j` – describing the initial configuration of the system, in particular the name of the MAS and the initial set of the agents that must be created and possibly some information and attributes that concern environment and agent implementation. The configuration file for the example is shown in Fig. 6, where ten instances of `producer` agents and ten instances of `consumer` agents are spawned. To launch multiple agents of the same type (e.g., ten producer agents) the cardinality can be specified as a parameter in the declaration (`#10`); the unique name of the agent in this case is given by the type and a progressive integer (in the example: `producer1`, `producer2`, etc).

By default, a single workspace called `default` is created and the specified agents are joined to this workspace. Actually a JaCa program can be composed by multiple workspaces and agents can concurrently join and work in multiple workspaces, either locally or in remote JaCa nodes. Workspaces can be created dynamically by agents by exploiting functionalities that are provided by a set of artifacts that are available, by default, in each workspace. Among the others, such a set includes: a `console` artifact, providing functionalities for printing on standard output; a `workspace` artifact, providing functionalities for managing the current workspace, including creating new artifacts (`makeArtifact` operation), disposing existing artifacts (`disposeArtifact`), discovering the identifier of existing artifacts (`lookupArtifact`), setting the security policies ruling the agent access to artifacts, etc.; a `blackboard` artifact, functioning as a blackboard – or better as a tuple space [29] – providing functionalities for enabling indirect communication and coordination among agents.

V. JaCa Programming: Further Features

In this section we focus on three main programming features among the others that are provided by JaCa, namely the capability of exploiting both direct communication based on message passing and indirect interaction through artifacts, the support for building distributed programs and the capability of integrating existing libraries, such as GUI toolkits. Further features are described in JaCa and CArtAgO technical documentation.

A. Integrating Direct Communication and Mediated Interaction

In JaCa agents can interact and communicate in two basic ways, either exchanging messages through speech acts [30] or by sharing and co-using artifacts functioning as interaction and coordination media [31]. The first way is generally referred as direct communication, while the latter as indirect or mediated communication. Both types of communication are important in programming concurrent and distributed programs, and we allow for exploiting them together.

The direct communication model is the one provided by the Jason language, based on a comprehensive subset of the KQML Agent Communication Language [30]. Among the available performatives, `tell` makes it possible to inform the receiver agent about some information (stored in the target agent as a belief), `achieve` to assign a new goal, and `ask` to request information. These performatives must be included in the communication action (`.send`) that actually sends the message, along with the specific parameters. An agent can react to the arrival of messages or, at a higher level, to the effect that the speech acts have, that are uniformly modeled as belief addition (for the `tell` performative) or goal addition (for the `achieve` performative).

To give a concrete taste of the approach, in the following we describe the realization of simplified version of the Contract Net Protocol (CNP) [32], in which both direct message passing and artifacts are used. In the example, a `ContractNetBoard` artifact (Fig. 11) is used by an announcer agent (code shown in Fig. 9) and five bidder agents (Fig. 10) to help their coordination in choosing the agent to whom allocate a task todo. Once the agent has been chosen, direct communication is used between the allocator of the task and the chosen agent to allocate the task and get the results.

Some brief explanation of the program behavior. In the main configuration file (Fig. 12), one announcer agent and five bidder agents are launched. The announcer opens the auction to allocate the task by performing an `announce` action over the `cnp_board` artifact (line 7). The artifact is observed and used also by the bidder agents, who are available for doing tasks. The `announce` action/operation executed by the announcer creates a new observable property `task_todo`, storing information about the new task (Fig. 11, lines 12-17).

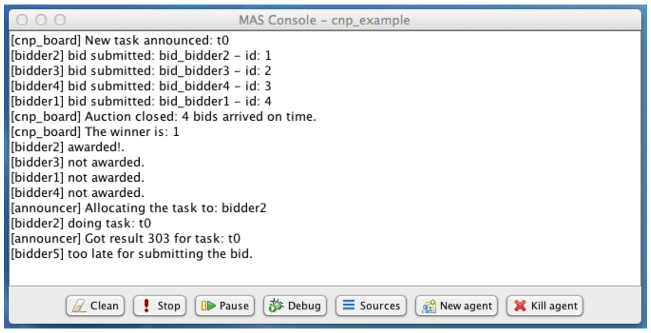
As soon as a bidder perceives that there is a new task to do, it reacts (Fig. 10, lines 10-22) by computing a new bid and issuing them on the contract net board by performing a `bid` action. The action can fail if the auction has been already closed by the announcer: in that case a message is printed on the console (lines 20-22). On the artifact side, the `bid` operation (lines 19-28) just adds the new bid to the list of bids received so far, if the auction is still opened, otherwise the operation fails (by executing the `failed` artifact primitive). As a detail, the third parameter of the `bid` operation is an action feedback parameter, i.e., an output parameter of the action bound to some value by the operation execution itself, set with a fresh identifier univocally identifying the bid.

The announcer waits some amount of time (2 seconds in the example), and then closes the auction by invoking the `close` operation (lines 8-9), which results in changing the `state` observable property of the artifact to `"closed"` and returns the list of information about the received bids as an action feedback parameter (lines 30-36). Such information are represented by instances of the `Bid` class. Then, the agent selects a bid (in the example the first one) and awards the bidder by performing an `award` action (lines 10-11), which results in updating the content of the `winner` observable property in the artifact (lines 38-41).

This change is perceived by bidder agents, which react in a different way depending on the fact that they are the winner or not (lines 24-28). After awarding, the announcer then communicates directly with the winner bidder by sending an achieve message specifying the task to be done (line 15). To retrieve the identifier of the bidder agent to whom sending the message, the method `getWho` is invoked on the selected bid object by means of the `cartago.invoke_obj` internal action.

Then, the awarded bidder reacts to the new goal to achieve (lines 35-37), just printing a message and then sending a message to inform the announcer about the task result (line 37). Finally the announcer reacts to the new belief communicated by the bidder (lines 19-20) by printing the result on the console.

A possible execution trace that can be obtained by launching the program is reported in Fig. 13, which shows the content of the Jason console. In that specific execution, four bidders were able to submit their bid on time and the winner was the bidder `bidder2` (whose bid identifier assigned by the `cnp_board` was 1).



```
[cnp_board] New task announced: t0
[bidder2] bid submitted: bid_bidder2 – id: 1
[bidder3] bid submitted: bid_bidder3 – id: 2
[bidder4] bid submitted: bid_bidder4 – id: 3
[bidder1] bid submitted: bid_bidder1 – id: 4
[cnp_board] Auction closed: 4 bids arrived on time.
[cnp_board] The winner is: 1
[bidder2] awarded!.
[bidder3] not awarded.
[bidder1] not awarded.
[bidder4] not awarded.
[announcer] Allocating the task to: bidder2
[bidder2] doing task: t0
[announcer] Got result 303 for task: t0
[bidder5] too late for submitting the bid.
```

Fig. 13. An execution trace of the CNP program, displayed on the Jason console.

```
/* announcer agent */

!allocate_task("t0",2000).

+!allocate_task(Task,Deadline)
  <- makeArtifact("cnp_board","ContractNetBoard",[]);
     announce(Task);
     .wait(Deadline);
     close(Bids);
     !select_bid(Bids,Bid);
     award(Bid);
     cartago.invoke_obj(Bid,getWho,Who);
     println("Allocating the task to: ",Who);
     .my_name(Me);
     .send(Who,achieve,task_done(Task,Me)).

+!select_bid([Bid|_],Bid).

+task_result(Task,Result)
  <- println("Got result ",Result," for task: ",Task).
```

Fig. 9. Source code of the announcer agent.

```
/* bidder agent */

task_result("t0",303).
!look_for_tasks("t0").

+!look_for_tasks(Task)
  <-  +task_descr(Task);
      focusWhenAvailable("cnp_board").

+task_todo(Task) : task_descr(Task)
  <-  !make_bid(Task).

+!make_bid(Task)
  <-  !create_bid(Task,Bid);
      .my_name(Me);
      bid(Bid,Me,BidId);
      +my_bid(BidId);
      println("Bid submitted: ",Bid," - id: ",BidId).

-!make_bid(Task)
  <-  println("Too late for submitting the bid.");
      .drop_all_intentions.

+winner(BidId) : my_bid(BidId)
  <-  println("awarded!.").

+winner(BidId) : my_bid(X) & not my_bid(BidId)
  <-  println("not awarded.").

+!create_bid(Task,Bid)
  <-  .wait(math.random(3000));
      .my_name(Name);
      .concat("bid_",Name,Bid).

 +!task_done(Task,ResultReceiver): task_result(Task,Res)
  <-  println("doing task: ",Task);
      .send(ResultReceiver,tell,task_result(Task,303)).
```

Fig. 10. Source code of bidder agents.

```
/* Contract Net Board artifact */

public class ContractNetBoard extends Artifact {
  private List<Bid> bids;
  private int bidId;

  void init(){
    this.defineObsProperty("state","closed");
    bids = new ArrayList<Bid>();
  }

  @OPERATION void announce(String taskDescr){
    defineObsProperty("task_todo", taskDescr);
    getObsProperty("state").updateValue("open");
    bids.clear(); bidId = 0;
    log("New task announced: "+taskDescr);
  }

  @OPERATION void bid(String bid, String who,
             OpFeedbackParam<Integer> id){
    if (getObsProperty("state").stringValue().equals("open")){
      bidId++;
      bids.add(new Bid(bidId,who,bid));
      id.set(bidId);
    } else {
      this.failed("cnp_closed");
    }
  }

  @OPERATION void close(OpFeedbackParam<Bid[]> bidList){
    getObsProperty("state").updateValue("closed");
    int nbids = bids.size();
    Bid[] vect = new Bid[nbids]; bids.toArray(vect);
    bidList.set(vect);
    log("Auction closed: "+nbids+" bids arrived on time.");
  }

  @OPERATION void award(Bid prop){
    signal("winner", prop.getId());
    log("The winner is: "+prop.getId());
  }

  static public class Bid {
    private int id;
    private String who, descr;

    public Bid(int id, String who, String descr){
      this.descr = descr; this.id = id; this.who = who;
    }
    public String getWho(){ return who; }
    public int getId(){ return id; }
    public String getDescr(){ return descr; }
    public String toString(){ return descr; }
  }
}
```

Fig. 11. Source code of the CNP board artifact.

```
MAS cnp_example {
  environment: c4jason.CartagoEnvironment
  agents:
    announcer agentArchClass c4jason.CAgentArch;
    bidder    agentArchClass c4jason.CAgentArch #5;
}
```

Fig. 12. Main configuration file of the CNP example, spawning one announcer agent and five bidder agents.

B. Distributed Programming and Open Systems Programming

JaCa intrinsically supports concurrent programming, in different ways: by exploiting Jason runtime architecture, agents are executed concurrently (and in parallel on a parallel HW, such as multi-core architectures); also, artifacts are executed concurrently, that is operations requested on different artifacts are executed concurrently.

Besides, JaCa directly supports also distributed programming: an agent running on some node can join workspaces that are hosted on a remote nodes, and then work with artifacts of the remote workspace(s) transparently. A simple example is shown in Fig. 14, in which an agent joins a remote `test` workspace located in `acme.org`, and, there, the agent prints some information on the console, creates a new `Counter` artifact called `c0` and uses it, by executing the `inc` operation and reacting to changes to the `count` observable property. While working on multiple workspaces, in JaCa a notion of *current* workspace is defined, being it the workspace implicitly referred when the agent invokes an operation over an artifact without specifying its full identifier. `current_wsp` is a predefined agent belief keeping track of current workspace. When an agent starts its execution, the current workspace is set by default to the `default` workspace. Then, it is automatically updated as soon as the agent joins other workspaces (including remote ones) or the agent executes a predefined `set_current_wsp` action. So, in the example, by joining the remote `test` workspace, this becomes the current workspace, and then the `println` action acts on the `console` artifact there, as well as the `makeArtifact` action that creates a new artifact overthere too. It is worth noting that in the plan reacting to a change to the `count` observable property (mapped on `count` belief), the agent prints a message on the console in the *original* workspace (lines 19-21): to disambiguate what console to use, in the action an annotation reporting the workspace where the artifact is stored is specified (line 21). The agent source code includes also a plan reacting to a failure in the plan handling the `!use_remote` goal, due to the fact that a `Counter` artifact called `c0` was already present in the remote workspace.

So in the overall this facility makes it possible to implement open systems with dynamic and distributed structure and behavior, given by the capability of agents of spawning other new agents dynamically, of joining dynamically existing workspaces or creating new ones, of creating / disposing artifacts belonging to a workspace. Given the distributed programming facility, a workspace can be joined by unknown agents of JaCa programs that have been spawned independently from the program where the workspace has been defined. The possibility of explicitly specifying security policies at a workspace level – by exploiting the functionalities provided by the `workspace` artifact – makes it possible to rule and govern such openness according to the need.

```
1  !test_remote.
2
3  +!test_remote
4    <- ?current_wsp(Id,_,_);
5       +default_wsp(Id);
6       println("testing remote..");
7       joinRemoteWorkspace("test","acme.org",WspID2);
8       ?current_wsp(_,WName,_);
9       println("hello there ",WName);
10      !use_remote;
11      quitWorkspace.
12
13 +!use_remote
14   <- makeArtifact("c0","examples.Counter",[],Id);
15      focus(Id);
16      inc;
17      inc.
18
19 +count(V)
20   <- ?default_wsp(Id);
21      println("count changed: ",V)[wsp_id(Id)].
22
23 -!use_remote
24         [makeArtifactFailure("artifact_already_present",_)]
25   <- ?default_wsp(WId);
26      println("artifact already created ")[wsp_id(WId)];
27      lookupArtifact("c0",Id);
28      focus(Id);
29      inc.
```

```
public class Counter extends Artifact {

  void init(){
    defineObsProperty("count",0);
  }

  @OPERATION void inc(){
    ObsProperty prop = getObsProperty("count");
    prop.updateValue(prop.intValue()+1);
  }
}
```

Fig. 14. An agent joining and working in a remote workspace *(top)*, and the source code of the counter used and observed remotely *(bottom)*.

C. Wrapping Existing Libraries and External Resources

Specific kind of artifacts can be designed and used to wrap and reuse existing libraries – written in Java but also in other languages, such as C and C++, exploiting the JNI (Java Native Interface) mechanism – making their functionalities available to agents, with a clean and uniform interface—which is the one provided by the artifact model. This allows in particular to build JaCa libraries that make it possible to access and interact with external resources existing in the deployment context or outside the system (such as a Web Services, a database, a legacy system).

A main example of JaCa library wrapping and integrating existing technologies is the one that allows for building and exploiting graphical user interface (GUI) toolkits. GUIs inside a JaCa program are modeled as artifacts mediating the interaction between humans and agents. A basic abstract artifact `GUIArtifact` is provided to be extended in order to create concrete GUIs. A GUI is designed then to make it observable to interested agents the events generated by the components (buttons, edit fields, list boxes,...) inside the GUI. Also, as an artifact, it provides operations that allow agents to interact with the GUI themselves, for instance to set the content of text fields.

Fig. 15 shows a simple example, in which an agent uses a GUI to repeatedly display the output of its work and to promptly react to user input. In particular, the agent creates

```
package c4jexamples;
...
public class View extends GUIArtifact {
  private MyFrame frame;

  public void setup() {
    frame = new MyFrame();
    defineObsProperty("value",0);
    linkActionEventToOp(frame.stopButton,"stop");
    linkWindowClosingEventToOp(frame, "close");
    frame.setVisible(true);
  }

  @INTERNAL_OPERATION void stop(ActionEvent ev){
    signal("stopped");
  }

  @INTERNAL_OPERATION void close(WindowEvent ev){
    signal("closed");
  }

  @OPERATION void setOutput(int value){
    frame.updateOutput(""+value);
    getObsProperty("value").updateValue(value);
  }

  class MyFrame extends JFrame {
    private JButton stopButton;
    private JTextField output;

    public MyFrame(){
      setTitle(".:: View ::.");
      setSize(200,100);
      JPanel panel = new JPanel();
      setContentPane(panel);
      stopButton = new JButton("stop");
      stopButton.setSize(80,50);
      output = new JTextField(10);
      output.setText("0"); output.setEditable(true);
      panel.add(output); panel.add(stopButton);
    }
    public void updateOutput(String s){
      output.setText(s);
    }
  }
}
```

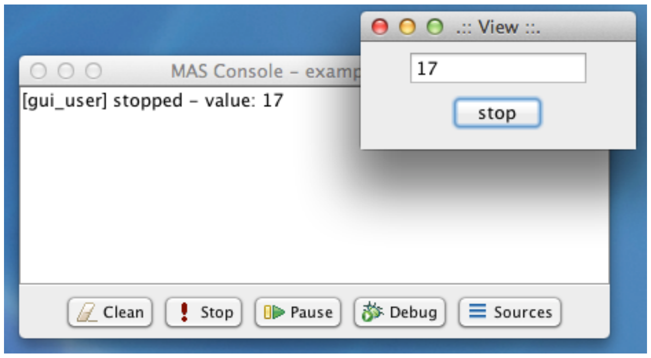
```
count(0).
!do_task_with_view.

+!do_task_with_view
  <-  makeArtifact("gui","c4jexamples.View",[],Id);
      focus(Id);
      !do_task.

+!do_task
  <-  -count(C);
      C1 = C + 1;
      +count(C1);
      setOutput(C1);
      !do_task.

+stopped : value(V)
  <-  .drop_all_intentions;
      println("stopped - value: ",V).

+closed
  <-  .my_name(Me);
      .kill_agent(Me).
```



```
[gui_user] stopped - value: 17
```

Fig. 15. Implementing and using GUI in JaCa: the View artifact *(left)*, the agent using the GUI *(right—top)* and the output of the program *(right—bottom)*.

a GUI artifact called View, providing one *stop* button and one output edit text. The structure of the GUI – based on Java Swing library – is defined by the MyFrame class, as it would be in a traditional OO program. An instance of this class is created inside View and events generated by the GUI components are linked to internal operations of the artifact by means of a set of predefined methods implemented in GUIArtifact. In particular an action event generated by frame.stopButton causes the execution of the internal operation stop, which generates an observable event stopped, and the window closing event is mapped onto the close operation, which generates a closed event. The agent first creates an instance called gui of the View artifact, and then repeatedly uses the view to display the results of its task, by means of the setOutput action (operation). While doing this task, the agent also observes the GUI and as soon as a stopped event is perceived, the agent reacts by suspending all its current ongoing activities (intentions) and printing in standard output a message. If a closed event is perceived, the agent terminates.

VI. Using JaCa in Real-World Application Contexts

We are currently applying the JaCa platform in different application domains, to stress the benefits but also the weaknesses of its programming model and more in general of the proposed agent-oriented programming approach.

One of these domains is the development of distributed applications based on Service-Oriented Architecture (SOA) and Web Services (WS) 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 SOA applications dynamically composing and orchestrating services [33], and from the foundational viewpoint, as a reference meta-model for the service-based approach, as suggested by the W3C Web Services Architecture reference document [34]. To this end, programming models and platforms are needed 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 co-operation. 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 characterizing 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 – e.g., interaction with existing Web Services (agents as service consumers), implementation of Web Services (agents as service providers) – eventually wrapping existing non-agent-oriented code. First results of this work are available in [35].

We are also investigating the adoption of our approach for the engineering of advanced Ambient Intelligence (AmI) applications. For the AmI context, a relevant research issue concerns how to concretely program non-intrusive applications exhibiting features such as context-awareness, personalisation, adaptivity and anticipation of users' desires [36]. To this end we applied our approach for realising a typical AmI application [37]: the management of a rooms allocation problem in the context of a smart co-working space – e.g., a school, an office building, etc. – where people can book and use rooms according to their needs and to the current occupancy schedule. The application has to set an autonomous and adaptive room management behavior in accordance with: *(i)* the events that are currently held – e.g., regulating the room temperature in accordance with the number of the event's participants, automatically turning off the lights for teaching events involving a projector, etc. – and *(ii)* also on the base of rooms (re)allocation in accordance with incoming user requests—i.e., aiming at optimising the number of events the system can host at any given time. Agents as usual encapsulate the control and decision-making part of the application, in this case related to monitoring and controlling facilities in rooms as well as deciding appropriate strategies to use for dynamic rooms allocation. The artifact-based distributed environment instead has been exploited to model and interface with the physical devices in the rooms (lights, temperature controllers, etc.), to model and represent high-level shared data structures with related operations (such as registers keeping track of room participants and schedules), besides typical coordination artifacts.

Another project where our agent-oriented programming approach has been applied concerns the engineering of an agent-based Machine-To-Machine (M2M) management infrastructure. M2M refers to technologies allowing the construction of automated and advanced services and applications (e.g., smart metering, traffic redirection, and parking management) that largely make use of smart devices (sensor and actuators of different kinds, possibly connected through a Wireless Sensor and Actor Network (WSAN)) communicating without human interventions. In [38] is discussed the realisation of an agent-based infrastructure to enable the deployment of city-scale M2M applications that share a common set of devices and network services. In such infrastructure each WSAN area

```
!init.

+!init
  <-  focus("NotificationManager");
      focus("SMSService");
      focus("ViewerArtifact").

+sms_received(Source, Message) : not (state("running"))
  <-  showNotification(Source, Message,
                 "jaca.android.sms.ViewerArtifact", Id).

+sms_received(Source, Message) : state("running")
  <-  append(Source, Message).
```

Fig. 16. Source code of the Jason agent that manages the SMS notifications.

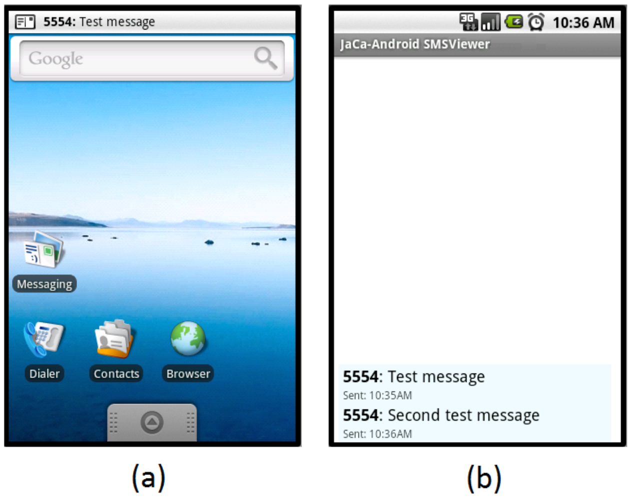


Fig. 17. The two different kinds of SMS notifications: (*a*) notification performed using the standard Android status bar, and (*b*) notification performed using the ViewerArtifact.

group is modelled through a CArtAgO workspace, where an agent acting as a *gateway* collects data sent from all the agents managing and controlling M2M devices (sensors, actuators) through artifacts. Finally a dynamic pool of agents regulate the functioning of each M2M infrastructural node in accordance with the current workloads experimented in the M2M infrastructure. The governance infrastructure is evaluated using a Smart Parking Management scenario, where an M2M system monitors the parking occupation in order to reduce traffic and to guide drivers through the streets.

The final domain we are considering in this paper is the engineering of smart mobile applications, in particular for pervasive and context-aware computing scenarios. To this end, JaCa has been ported on the Android platform [39], enabling the development of Android applications using agent-oriented programming [40] [41]. The project is called JaCa-Android. 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, Fig. 16 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 `SMSService` 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` 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 `NotificationManager` artifact is used to show messages on the Android status bar, providing a `showNotification` operation to this end. Depending on 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 1-6), is governed by two reactive plans. The first one (lines 8-10) 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 (Fig. 17, *(a)*). The second plan instead (lines 12-13) 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 (Fig. 17, *(b)*): this is done by executing the `append` operation provided by `ViewerArtifact`.

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 design and program the business logic of the applications by suitably defining the Jason agents' behavior, and she only need to acquire the ability to work with the artifacts that are specific of the new application context.

VII. Towards a New Generation of Agent-Oriented Programming Languages for Computer Programming

By exploiting existing agent technologies (Jason and CArtAgO in particular), JaCa makes it possible to concretely experiment agent-oriented programming as a general-purpose paradigm for computer programming and software development, getting in practice some of the benefits of agent-orientation described in Section III. However, the approach lacks of some fundamental features when compared to current languages for software development – such as the object-oriented ones – due to the fact that the agent programming models / technologies on which JaCa is based have been designed having Distributed Artificial Intelligence problems in mind, not software development in general. These missing features concern desiderata that are not crucial from an AI point of view, but from a software engineering and programming perspective.

A first important desideratum concerns *error checking*, i.e., the possibility to detect errors in programs before executing them. Not only syntax errors, but also errors concerning the semantics of the program: examples are allocating tasks to agents that have not plans to handle them, or executing actions that are not part of the interface of an artifact, or rather having agent plans that react to events that are never generated by the artifacts used in the plans. To this purpose, current AOP languages offer very limited and ad-hoc capabilities. In programming languages and software engineering, this issue is addressed by introducing a sound notion of *type* and *type systems* [42]. So designing agent-oriented programming languages with strong typing would allow for type checking programs at compile type, strongly impacting on the process of program development.

Typing is important also for the program organization and as a conceptual tool for building more clean and elegant systems. In fact, the definition of a notion of *subtyping* is the base for introducing *conceptual specialization* in program organization, and then defining a *substitutability principle* [43] also in agent-oriented programs, getting finally a safe way to extend and reuse program specifications.

Inheritence instead [43] – along more recent mechanisms such as *traits* [44] – are important features in OOP to achieve code reusability and a way to define hierarchies and compositions that relate implemented parts of a systems (such as classes). So we believe that suitable mechanisms that foster code reuse are important also for the agent-oriented paradigm, both on the agent side – for instance, making it possible to define new agents from existing ones, inheriting their capabilities (such as their plans) – and on the environment side – for instance, defining the artifact classes by extending existing ones, so inheriting their operations and observable properties.

Besides typing and inheritance, a stronger support for *modularity* [45]. Finding suitable abstractions and mechanisms to improve the modularization of agent behavior is a main issue also in current research in agent programming languages (examples are [46], [47], [48], [49], [50], [51]). In the case of Jason for instance, the source code inside an agent to achieve some goal is fragmented into a flat sequence of typically small plans, that trigger each other. A notion of module – similar to the notion of *capability* [47] adopted by the JACK platform [17]– has been recently proposed to improve Jason agent modularity [51]. Actually, among all the possible solutions that can be adopted for achieving a good modularity, we are interested in those that allow for contextually introducing also mechanisms for reuse such as inheritance and for keeping a strong separation between specification of the behaviors (through typing) and their (hidden) implementation.

Finally, we argue that a modern programming language designed for software development in general must necessarily have a good or seamless integration with object-oriented and functional programming, that are very strong and mature paradigms for defining and working with data structures and related purely transformational algorithms. This is not the case for existing agent programming languages, that are typically based on logic programming and do not provide a seamless and efficient support to manipulate objects.

These motivations lead us to explore the definition and development of new agent-oriented programming languages, integrating and embedding in a sound way all these features from their foundation. This is the objective of the simpAL project, whose first results are reported in [52]. simpAL is an agent-oriented programming language designed from scratch so as to embed some main ideas and features of JaCa, but taking object-oriented programming and in particular Java-like languages as reference for defining and manipulating data structures, and to integrate features like an explicit notion of typing and inheritance. Actually simpAL is not meant to replace JaCa, which we consider the reference platform–along with JaCaMo [53], which extends JaCa to support also organization-oriented programming—for exploring the development of multi-agent systems for tackling problems in Distributed Artificial Intelligence contexts.

VIII. 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. In the state-of-the-art, agents and multi-agent systems have been explored so far mainly as an approach for tackling AI and DAI problems: in this paper we solicited a further perspective, which aims at exploring the value of agent-orientation as a programming paradigm, providing an effective level of abstraction to tackle the complexities which characterize modern programming (e.g, concurrency). In order to show in practice some of the main concepts underlying the approach, we exploited the JaCa platform, which is based on existing agent-oriented technologies—the Jason language to program agents and CArtAgO framework to program the environment. JaCa technology can be used to concretely experiment the approach for developing real-world applications tackling some of the main aspects that characterize today software system complexity, such as concurrency, distribution, reactivity, flexibility and autonomy. Finally, we shed a light on some fundamental features that are missing today in existing agent programming technologies and languages (such as typing and inheritance), which can be considered a must-have for us to investigate agent-orientation as a general-purpose paradigm for computer programming and software development. Future work will be devoted in particular to both verify the effectiveness of the approach in practice, using agent-oriented programming to tackle relevant programming problems and projects, and to improve our current models and technologies—JaCa and simpAL, in particular.

References

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

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

[3] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 2009.

[4] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue: Tomorrow's Computing Today*, 3(7):54–62, September 2005.

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

[6] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Special Issue: Multi-Agent Programming*, volume 23 (2). Springer Verlag, 2011.

[7] Rafael H. Bordini, Mehdi Dastani, and Amal El Fallah Seghrouchni, editors. *Multi-Agent Programming Languages, Platforms and Applications - Volume 1*, volume 15. Springer, 2005.

[8] Rafael H. Bordini, Mehdi Dastani, Amal El Fallah Seghrouchni, and Jürgen Dix, editors. *Multi-Agent Programming Languages, Platforms and Applications - Volume 2*. Springer, 2009.

[9] Alessandro Ricci and Andrea Santi. Agent-oriented computing: Agents as a paradigm for computer programming and software development. In *Proc. of the 3rd Int. Conf. on Future Computational Technologies and Applications (Future Computing '11)*, pages 42–51, Rome, Italy, 2011. IARIA.

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

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

[12] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *1st Int. Conf. on Multi Agent Systems (ICMAS'95)*, 1995.

[13] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

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

[15] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17 (3), December 2008.

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

[17] Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. JACK intelligent agents^{TM} — summary of an agent infrastructure. In *Proc. of 2nd Int. Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[18] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI reasoning engine. In Rafael Bordini, Mendi Dastani, Jurgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming*. Kluwer, 2005.

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

[20] Michael David Travers. *Programming with Agents: New metaphors for thinking about computation*. Massachusetts Institute of Technology, 1996.

[21] Bonnie Nardi, editor. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, 1996.

[22] David Kirsh. Distributed cognition, coordination and environment design. In *Proceedings of the European conference on Cognitive Science*, pages 1–11, 1999.

[23] Alessandro Ricci, Andrea Omicini, and Enrico Denti. Activity Theory as a framework for MAS coordination. In Paolo Petta, Robert Tolksdorf, and Franco Zambonelli, editors, *Engineering Societies in the Agents World III*, volume 2577 of *LNCS*, pages 96–110. Springer, April 2003.

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

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

[26] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

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

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

[29] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[30] Y. Labrou, T. Finin, and Yun Peng. Agent communication languages: the current landscape. *Intelligent Systems and their Applications, IEEE*, 14(2):45 –52, mar/apr 1999.

[31] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. Coordination artifacts: Environment-based coordination for intelligent agents. In *Proc. of the 3rd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, volume 1, pages 286–293, New York, USA, 19–23July 2004. ACM.

[32] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comput.*, 29:1104–1113, December 1980.

[33] Michael N. Huhns, Munindar P. Singh, and Mark et al. Burstein. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):69–70, November 2005.

[34] W3C Web Service Architecture – http://www.w3.org/TR/ws-arch/, last retrieved: June 21th 2012.

[35] Alessandro Ricci, Enrico Denti, and Michele Piunti. A platform for developing soa/ws applications as open and heterogeneous multi-agent systems. *Multiagent Grid Syst.*, 6:105–132, April 2010.

[36] P. Remagnino and G. L. Foresti. Ambient intelligence: A new multidisciplinary paradigm. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 35(1):1–6, 2005.

[37] A. Sorici, O. Boissier, G. Picard, and A. Santi. Exploiting the jacamo framework for realising an adaptive room governance application. In *Proc. of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, pages 239–242. ACM, 2011.

[38] C. Persson, G. Picard, F. Ramparany, and O. Boissier. A jacamo-based governance of machine-to-machine systems. In *Proc. of the 10th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 12)*, Advances in Soft Computing Series. Springer, 2012.

[39] Android Platform web site – http://www.android.com/, last retrieved: June 21th 2012.

[40] Andrea Santi, Marco Guidi, and Alessandro Ricci. JaCa-Android: An agent-based platform for building smart mobile applications. In M. et al. Dastani, editor, *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, volume 6822 of *LNAI*, pages 95–119. Springer, 2011.

[41] JaCa-Android project web site – http://jaca-android.sourceforge.net/, last retrieved: June 21th 2012.

[42] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–523, December 1985.

[43] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '88, pages 55–77, London, UK, UK, 1988. Springer-Verlag.

[44] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28:331–388, March 2006.

[45] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[46] M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Ch. Meyer, and Frank S. de Boer. Goal-oriented modularity in agent programming. In *Proc. of the 5th Int. Joint Conf. on Autonomous agents and Multiagent systems (AAMAS'06)*, pages 1271–1278, New York, NY, USA, 2006. ACM.

[47] P. Busetta, N. Howden, R. R *o*nnquist, and A. Hodgson. Structuring BDI agents in functional clusters. In N.R. Jennings and Y. Lespèrance, editors, *Intelligent Agents VI*, volume 1757 of *LNAI*, pages 277–289. Springer, 2000.

[48] L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the capability concept for flexible BDI agent modularization. In *Programming Multi-Agent Systems*, volume 3862 of *LNAI*, pages 139–155. Springer, 2005.

[49] Peter Novák and Jürgen Dix. Modular BDI architecture. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1009–1015, New York, NY, USA, 2006. ACM.

[50] Koen Hindriks. Modules as policy-based intentions: Modular agent programming in GOAL. In *Programming Multi-Agent Systems*, volume 5357 of *LNCS*, pages 156–171. Springer, 2008.

[51] Neil Madden and Brian Logan. Modularity and compositionality in Jason. In *Proceedings of International Workshop Programming Multi-Agent Systems (ProMAS 2009)*. 2009.

[52] Alessandro Ricci and Andrea Santi. Designing a general-purpose programming language based on agent-oriented abstractions: the simpal project. In *Proc. of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, VMIL'11*, SPLASH '11 Workshops, pages 159–170, New York, NY, USA, 2011. ACM.

[53] Olivier Boissier, Rafael H. Bordini, Jomi F. Hbner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, (0):–, 2011.