Designing a General-Purpose Programming Language based on Agent-Oriented Abstractions: The simpAL Project

Alessandro Ricci
University of Bologna, Italy
a.ricci@unibo.it

Andrea Santi
University of Bologna, Italy
a.santi@unibo.it

Abstract

The fundamental turn of software toward concurrency, decentralization, distribution, interaction calls for conceptually extending or evolving mainstream programming paradigms with proper high-level features to tackle these aspects. To this purpose, in this paper we discuss the value of *agent-oriented programming* as a general-purpose programming paradigm to tackle this challenge, and, in particular, we present an agent-oriented programming language called simpAL, which is based on agent-oriented abstractions that are meant to simplify the programming of modern applications.

Categories and Subject Descriptors D.1 [*Programming Techniques*]; D.2 [*Programming Languages*]

General Terms Languages, Design

Keywords agent-oriented programming, actors

1. Introduction

The fundamental turn of software toward concurrency, decentralization, distribution, interaction that we are witnessing in recent years calls for conceptually extending or evolving mainstream programming paradigms with proper high-level features to tackle these aspects. As stated by Sutter and Larus in [18], *the free lunch is over*. Actually the free lunch is over not only related to concurrency, but also to distribution, decentralization, reactivity – in the sense of reactive systems as defined in [9] – also autonomy. Besides introducing fine-grain mechanisms or patterns to exploit parallel hardware and improve programs efficiency in existing mainstream languages, it is now increasingly important to introduce higher-level programming abstractions that "help build concurrent programs, just as object-oriented abstractions help build large component-based programs" [18].

In that perspective, our general research aim is to explore *agent-oriented programming* as a high-level general purpose programming paradigm providing a proper set of abstractions that are meant to simplify the design and programming of concurrent, distributed, reactive and interactive programs.

Actually, the idea of agent-oriented programming is not new, being introduced by Shoham in 1993 [17] and since then many agent and multi-agent programming languages have been proposed in literature [4, 6]. However, the focus of these works so far have been mainly on architectures, theories, languages to program agents and multi-agent systems in a (D)AI context, putting most of the effort and emphasis on theoretical and practical issues relevant for that context. Given the AI background, current agent-oriented programming languages in the state of the art (briefly accounted in Section 7) are based on logic programming, focusing on features that are especially important in that context – such as providing some basic reasoning capability. This paper instead proposes agent-oriented programming as a general-purpose programming approach, as an evolution of the object-oriented and actor based ones, focusing then on those aspects that are important from a programming and software development perspective, and finally investigating its value for the *free lunch is over* call. In particular we are interested in: *(i)* identifying the essential concepts and features of the paradigm, and investigating how such features could be effective in particular for tackling complexities of modern software programming; *(ii)* investigating how well-known features and mechanisms that have been introduced and developed in modern programming languages to support programming in the large and good programing (but in the sequential case), could be injected and eventually reframed by adopting an agent-level of abstraction. An example is typing: is it possible to define a type system also for agent-oriented abstractions so as to, e.g., have strong error-checking at compile time? Is it possible to define a subtyping relationship also for, e.g., autonomous agents so as define and exploit some kind of substitution property? An other example concerns code reuse and inheritance, so introducing concepts, theory and mechanisms that allow for extending, e.g., agent behaviour incrementally. Finally, *(iii)* investigating if and how the new abstraction level raised by agent-oriented programming impacts on the design of tools supporting the development and deployment process, from front-end to debuggers and profilers.

The method we chose to explore these points is the design and development of new programming language called simpAL, and related platform / infrastructure and tools. On the one side, differently from agent-oriented programming languages in the DAI context, simpAL is not based on logic programming but on object-oriented programming to define data structures and related algorithmic parts. On the other side, by taking inspiration from existing agent languages and frameworks, it adopts the Belief-Desire-Intention (BDI) [14] as reference background model/architecture for defining agents and the A&A conceptual model [12, 15] to define the agent environment as first-class design and programming concept.

In this paper we present first results about the design and development of simpAL. First, we provide an overview of the main concepts of the language (Section 2), and then we go into the details of the features of the main abstractions of the language – namely the agent abstraction (Section 3), the environment abstraction (Section 4) and the organization abstraction (Section 5). Finally, we conclude the paper by discussing related work (Section 7) and sketching a roadmap of our future work (Section 8).

2. simpAL: Overview of the Main Concepts

Quoting Lieberman [11], *"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"*. Following this perspective, agent-oriented programming can be framed as an evolution of object-oriented and actor-based 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 particular, the inspiration for conceiving the features of simpAL abstractions – at the root of its computation and programming model – comes from human organizations and cooperative working environments.

2.1 Agents, Artifacts and Workspaces

A program in simpAL is conceived like a human organization where articulated concurrent and coordinated activities take place, distributed in time and space, by *agents* - i.e., the members of the organization – working in a common environment, organized in *workspaces* (see Figure 1). Activities are explicitly targeted to some objectives. The complexity of work calls for some division of labor, so each member is responsible for the fulfillment of one or multiple *tasks*, by virtue of its *role* inside the organization. So agents in simpAL are active components in charge of performing

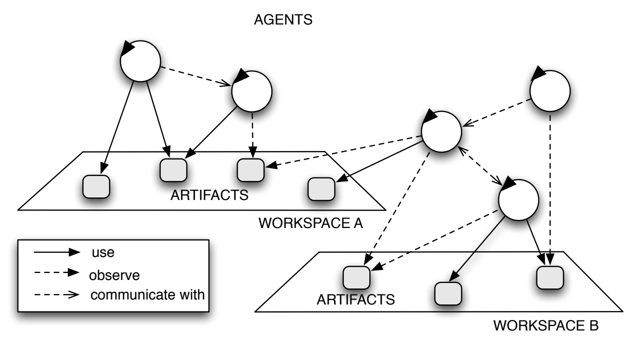

Figure 1. Abstract view of a simpAL program.

autonomously some task. Autonomously means in this case that given a task to do, they pro-actively decide what actions to do and when to do them, promptly reacting to relevant events from their environment, fully encapsulating the control of their behaviour.

Interaction is a main dimension, due to the dependencies among the activities. Cooperation occurs by means of both direct message-based communication – like in the actor model [1] – and through resources and tools composing the shared environment, represented by the *artifact* abstraction. An example of artifact is a shared blackboard, but also a simple counter, a clock, or rather a database—so artifacts can be used to directly represent any computational entity that conceptually does not need to be autonomous, but that can provide functionalities that can be exploited by agents. Analogously to artifacts in human organizations, artifacts in simpAL can play an essential role to support indirect form of coordination or to explicitly represent the co-constructed results of workers' activity. Like artifacts in the human case, artifacts in simpAL can be dynamically created (by agents) and disposed, and eventually can be designed to be composed, so as to create complex artifacts by connecting simpler ones.

Agents interact with other agents by means of *communication actions*, which allow to asynchronously exchange messages making it explicit the objective of the message, for instance to inform about something or to assign a task. Agent-artifact interaction is based instead on the concept of *use* and *observation*, reminding the way in which artifacts are used by people in human environments. In order to be used, artifacts provide some set of *operations*, that correspond to the set of actions available to agents to use them. So the repertoire of an agent's actions at runtime depends on the artifacts that the agent knows and can use. Besides operations, the usage interface of an artifact includes also *observable properties*, as observable information concerning the dynamic state of the artifact which may be perceived and exploited by agents accordingly. In this way, the observer pattern and event-oriented communication are directly supported by the core abstractions of the model (and language), but without inversion of control—this will be clarified later on.

Finally, the environment of an organization can be structured into one or multiple *workspaces*, possibly distributed in multiple nodes of the network. A workspace is a logical container of artifacts and finally of the activities occurring inside the organization; it is an abstraction useful then to define the logical topology of the organization, which can be distributed, and a related notion of locality. An agent of the organization can concurrently join and work transparently in multiple workspaces.

2.2 First General Remarks

First, the approach promotes a *decentralized mindset* in programming: there is not a unique locus of control in the system, which is decentralized instead into agents (deciding which actions to so and when) and artifacts (hosting action execution). The decentralization is first of all logical: the number of physical threads used to run a (possibly distributed) program is related to the number of processors or core available in the system, not the number of agents/artifacts to be executed. Such a support to decentralized mindset is in common with other main approaches in concurrent programming, starting from the actor model [1]. Compared to actors, here we don't have a single abstraction to represent every component of the system but two dual abstractions: one conceptually representing autonomous entities (agents) and one representing non autonomous entities that are used and observed by the autonomous one (artifacts). On the one side, the drawback for this choice is breaking uniformity, on the other side this allows for raising the level of abstraction. In actor-based approaches – including those based on processes like Erlang [2] – everything must be an actor, so also those entities that conceptually are not thought to be autonomous or exhibiting a pro-active behavior. An often cited example is the bounded-buffer in producer-consumer architectures. It is more natural to model such an entity as an artifact, providing actions to insert and remove items, possibly coordinating such actions so as to avoid interferences and to synchronize producers and consumers agents, than as autonomous entity exchanging asynchronous messages.

Then, a natural question is: where are objects? Are they replaced by artifacts? The answer is no, objects are used to define the data model of programs. That is, agents and artifacts as meant to be used as coarse grain abstractions to define the shape of the organization (that is, of the program), in particular of the control part of it (decentralized, distributed). This layer is fairly independent from the paradigm and language adopted to represent data structures and purely transformational computation. In the case of simpAL to this end we adopted an object-oriented programming language, in particular a subset of Java, that is the pure OO part of the language, excluding constructs and mechanisms introduced for concurrency. So objects are the basic data structures used inside agents, artifacts and in their interactions.

3. The Agent Abstraction

The model/architecture adopted for agents in simpAL is a simplification of the Belief-Desire-Intention (BDI) one [14], yet extended with features explicitly devised with programming and development in mind. This choice is motivated by the capability provided by that model to represent active components that need to integrate both task-oriented / pro-active and even-driven / reactive behaviors, providing also an effective way to modularize complex behaviour.

In the following, first we provide a bird's eye view on the model, then we describe in more detail some key concepts, providing also some snippets of simpAL code to exemplify the description.

3.1 Overview

An agent is a state-full *task-* and *event-driven* entity, able to pursue autonomously tasks according to the *role(s)* it implements inside the organisation. Events concern both changes in the environment and messages sent by other agents. The private memory of the agent is given by a memory store called *belief-base*, in which the information that the agent has about its private state, about the observable state of the environment and about information communicated by other agents is collected in terms of variable-like information items called *beliefs*. To perform tasks, the agent exploits the *plans* available in its *plan library*. Plans are modules of procedural knowledge specifying how to act and react to environment events in order to accomplish some specific task. The set of plans in the plan library depends on the *scripts* loaded by the agent, which group set of related plans to play some specific role(s).

3.2 Agent Beliefs and the Belief-Base

Structurally, a belief in simpAL is similar to a variable in procedural languages, being defined by a name, a type and a value. Beliefs are used uniformly to represent information that the agent knows, which can actually have a different source. First, beliefs can refer to some information defined by the agent itself, in the agent program, either in scripts or in plans. In this case beliefs are analogous to instance fields of classes and local variables declared in methods; they are defined at compile time, and they can be accessed by name both to read the value and assign new values.

Then, beliefs can refer to the observable state of an artifact and the source in this case is the artifact itself. As will be detailed in Section 4, the observable state of an artifact is represented by a set of variable-like *observable properties*: when using an artifact, on the agent side automatically a belief is created for each observable property of the artifact and the value is automatically updated as soon as the

```
role MyRole extends BasicRole {
  task MyTask {
    aParam: int; // primitive type
    myList: List<String>; // an object
    myCounterTool: Counter; // an artifact
  }
  task AnotherTask {...}
}
```

Figure 2. Role definition in simpAL.

change is perceived by the agent[1]. Differently from the previous case, beliefs about observable properties are dynamic, in the sense that the actual set of beliefs of this kind depends on the set of artifacts (and the observable properties) that the agent is using (observing). This is explicitly declared in plans, by means of an attribute called `using`, as will be described later. The belief is automatically removed as soon as an agent is no more using the artifact.

Finally, beliefs can refer to information communicated by some other agent through a specific communication action (*tell*). In that case the source of the belief is the sender agent and beliefs are dynamic: a new belief is dynamically created for each new information communicated by other agents by means of specific communication acts. The set of possible beliefs sent by means of a tell that an agent can understand is explicitly declared in task definition.

It is worth remarking that in existing agent-oriented languages beliefs are typically represented by first-order logic literals, denoting information that can be used by reasoning engines. However the logic representation is not necessarily part of the belief concept, as remarked by Rao and Georgeff in [14][2].

3.3 Tasks and Roles

The notion of task is used pervasively in the analysis and design of concurrent and distributed programs, as a starting point to decompose and modularize the specification of what a system has to do. For *task* here we mean any kind of description of some well-defined unit of work to be done—so *what* must be done, not how to do it. In simpAL such a concept is brought into the language level, so as to explicitly represent and specify what drives the agent active and autonomous behavior. In fact, an agent starts doing something, selecting autonomously what actions to do, if and only if it has at least some task to accomplish—which can be assigned to the agent also statically, at boot time. The notion of *role* is introduced to explicitly group a set of tasks, defining then explicitly what are the tasks that agents implementing a role are supposed to be able to do.

[1] during the execution cycle, described in Section 3.7

[2] *"[beliefs] can be viewed as the informative component of the system state"* and *"[beliefs] may be implemented as a variable, a database, a set of logical expressions, or some other data structure"*([14], p. 313)

Figure 2 shows the definition of a role and of a set of tasks. A role is identified by a name and the definition of the tasks. Each task is defined by a name, a set of parameters – representing information about the task to do or done, specified by agents at runtime – and a set of predefined optional attributes, that allow for specify further information about the task, such as the expected information that can be told to agents performing the task (understands attribute), the goal of the task i.e. the condition that the task aims at making true (goal attribute), the list of tasks that must be accomplished before this one (pre attribute).

It is worth noting that tasks are not a first-class concept in actors and (concurrent) object-oriented systems, which are based instead purely on messages (and methods). In that case tasks are implicit: an object assigns or delegates a task to another object by sending it a message. At the same time, messages in actors and processes in particular – as well as in agents – can be used not only to ask to do some task, but also simply to communicate some information.

Introducing an explicit notion of task distinct from messages allows to define more clearly the concept of *pro-active* behaviour, distinguishing it from a *re-active* one. Actors (and objects) are often labelled as reactive since the only way for an actor to do something is by reacting to the receipt of a message. Agents – as defined here – are pro-active since they move and keep moving as soon as they know to have a task to accomplish, so as achieve the task – independently from how the task have been assigned to them (not necessarily by means of receiving a message). And they keep moving until the task has been accomplished (or the task is aborted because of some failure).

3.4 Typing Agents with Roles

We use the notion of task to define a notion of type for agents. In the case of objects/actors, the type defines the set of messages that an object/actor of that type can accept, in spite of its implementation. The type of an agent is given then by the role or the set of roles that it declared to implement, independently from the concrete behavior that it will exhibit to play the role(s) and accomplish the tasks.

Analogously to the object-oriented case, the availability of well-defined notion of type also for agents allows to: *(i)* improve error checking at compile time, i.e. if an agent declares to implement some role, then at the implementation level it must implement the necessary to achieve every task specified in the role (more about this in next subsection); *(ii)* define and exploit a sub-typing relationship among roles, to make it easier the incremental development and reuse of programs. In particular, a role R' declared as an extension of a role R implicitly includes all the tasks defined in R or their extension/refinement, and possibly the definition of new tasks. In that way, an agent who declares to play the role R' can substitute any agent who plays the role R.

As a final note, the notion of role has a value also at the *organizational level* as well (described in Section 5), allowing for explicitly specifying information about the overall "social" structure of the system (organization), in terms of the possible roles available and related relationships.

3.5 Plans and Scripts

A task describes what has to be done, not *how* to do it. To this end we introduce a notion of *plan* as a construct to encapsulate the procedural knowledge that an agent can re-use and exploit in order to fulfill some task, and finally to define any block of agent behaviour. The term *procedural knowledge* is referred to the fact that the content of the plan is used by the agent to select the sequence of actions to do in order to fulfill some task—and, as will be described in next subsection, in the simplest case this is analogous to a simple procedure.

The set of the plans of an agent is stored in the agent *plan library*. Actually also the content of the library is represented also in the belief base, that is an agent "knows" what plans it has. The overall behavior of an agent is then defined by the set of plans that it has in its plan library. So from a programmer point of view, programming an agent means specifying a set of plans to be included in agent plan library. At runtime, given a task to do, the agent selects from the plan library an *applicable* plan for the task and starts executing it. Conceptually, this is analogous – at a first glance – to message dispatch in objects/actors, where a message (task) is handled by executing the corresponding method (plan).

Actually, for the same type of task, an agent could have multiple plans that can be used in different *context*. The context is defined by the agent belief base, which includes also beliefs about the current value of the task to be executed. Thus, the definition of a plan can include – besides the attribute specifying explicitly the type of the task for which the plan is good for – also a context attribute, specifying the condition (as a boolean expression over the belief base) in which the plan can be used. So, by refining what previously said, at runtime a plan is selected if type of task specified in the attribute is compatible with the type of the assigned task and if the condition expressed in the context attribute holds, given the current state of the agent belief base. This makes it possible then to modularize the agent behavior not only on the basis of tasks, but also on the runtime context of the agent. This is a nice feature in particular to simplify the implementation of context-aware behavior.

In order to group and modularize set of plans related to the same task or to the tasks of the same roles, the notion of *script* is introduced. Conceptually, an agent script describes *how* to play one or more *roles*, as defined previously. More practically, it contains the definition of the set of plans which can be exploited by an agent to play some role, and possibly also the definition of the some global beliefs that are shared and accessed in plans. Figure 3 shows an abstract example of agent script in simpAL, including only the skeleton of the plans. Beliefs are declared specifying the name of the belief, the type of the value of the belief and optionally the initial value. Types range from primitive data types, object-oriented (Java) data types (so interfaces, classes) or simpAL specific types (such as roles and artifact usage interfaces).

```
agentscript MyScript implements MyRole {
  /* long-term beliefs */
  value: double;
  st: String;
  myTool: MyTool;

  plan MyPlanA
    task: MyTask
    context: (aParam > 0 && st.equals("test")) {...}

  plan MyPlanB
    task: MyTask
    context: aParam <= 0
    using: myCounterTool, myTool {...}

  plan MyPlanC
    task: AnotherTask  {...}
  ...
}
```

Figure 3. Script definition in simpAL.

In plan definition, task: and (optionally) context: are attributes used to specify respectively the task for which the plan can be used and the context. The expression in context: can refer also to the actual value of the parameters of the task specified in task:, that triggered the selection/execution of the plan (e.g., aParam). Other attributes are useful for declaring information about the plan execution, for instance: declaring explicitly the list of the artifacts that are going to be used (and observed) in the plan, such as in the the plan MyPlanB in Figure 3 (using: attribute); declaring explicitly the condition over the belief-base to consider the plan completed (goal attribute); declaring if the plan should be executed atomically (atomically: attribute) – that means as a single action.

So, analogously to classes in OO, agent scripts are useful to provide a form of *encapsulation* and *information hiding* in agent implementation: they bundle together beliefs as informational state and plans as procedural knowledge, making beliefs only accessible internally to plans. Plans instead are an explicit mechanism to *modularize* agent overall behavior, providing also a disciplined way to allow for an agent to possibly extend its behavior at runtime, by creating and adding plans to the plan library.

3.5.1 Error Checking, Extension and Polymorphism

If roles are what defines the interfaces for agent, agent scripts define their implementation and they are kept separated. This makes it possible to do some error checking on agent script at compile time: If an agent script implements some role R, then it must define at least one plan for each task defined

```
plan MyPlan task: MyTask, using: myCounterTool,
                                  console @ main {
  info: int;
  friend: RoleB;
  meetingTime: long;
  meetingRequest: MeetingInfo;
  ...
  changed count : count > 0 => someAction1  #act
  when done_act => inc (delta: 2) on myCounterTool
  told ping by friend => println (msg: "news")
  changed time : time == meetingTime
               => tell meetingRequest to: friend
  always => info = info + 1
  always : info < 100 => info = info + 1

  changed count : count < 0 =>  {
       inc (delta: 2) on myCounterTool
       println(msg: "inc done")
  }
}
```

Figure 4. A plan definition with some action rules.

in R. Otherwise, it means that the script does not provide enough skills to play the role and an error is raised.

Moreover, as in the case of classes in OOP, scripts make it possible to define an extension and reuse mechanism for agent for what concern the behavior/plan level, to have some support to incremental development. In particular, an agent script S' can be defined as an extension of an agent script S, inheriting in that case all the plans and beliefs defined in S (problems related to name clash are managed accordingly) and possibly adding further plans either to achieve tasks already part of the role implemented by S, or new tasks defined in a role R' implemented by S' (and not by S).

Finally, this characterization of scripts and plans along with the notion of task and role defined previously make it possible to define a well-defined notion of *polymorphism* also for agents: *(i)* the same task can be performed by different agents implementing the same role in different ways, depending on the scripts that they are using and then the set of plans described in scripts; *(ii)* a task assigned to an agent can be performed in a different way depending on the contextual conditions.

3.6 Plan Body: Actions Rules

Integrating both active and reactive behavior is an important and hard programming issue, more generally related to the well-known problem of integrating thread-based and event-based systems [8]. In simpAL this point is tackled by the model adopted defining the procedural knowledge of plans, which has been conceived in order to allow for easily specifying both sequence or workflows of actions but also *reactions* to events perceived asynchronously by the agent.

A plan body is given by a set of *local beliefs* and *action rules*. Local beliefs are beliefs whose temporal and lexical scope is given by the plan body, analogous to local variables in procedures. So they are automatically added to the belief base when a plan is instantiated and removed when a plan in execution completes or aborts. The syntax to declare a local belief is the same as for the beliefs declared at the script level.

Action rules specifying *what* action to do and *when*. Similarly to Event-Condition-Action (ECA) rules, *when* is the defined by the description of an event description E and a condition C as a boolean expression over agent beliefs, which include internal beliefs and beliefs about the state of the environment. In the most general case, an action rule is syntactically represented as E : C $\Rightarrow$ A [label]. The informal semantics is: if an event matching the E description occurs and contextually the condition C holds, then select the action A to be executed and identify it with the `label` identifier. Then, a plan in execution is considered completed as soon as: *(i)* the goal attribute specified in the plan or in the corresponding task is satisfied; *(ii)* in the current plan there are no action rules that can be triggered from now on (because no events specified can happen anymore).

3.6.1 Events

Events concern changes to agent beliefs, caused by either (1) some percepts received from the environment, or (2) messages received by other agents, or rather (3) the passing of time.

The first case concerns changes to beliefs about the current value of the observable properties of artifacts that are used by the agents in the plan. The syntax defining the event in this case is changed *ObsBel*, where *ObsBel* is the name of the belief corresponding to the name of the observable property of an artifact, possibly including also the identifier of the artifact, source of the belief. In the example showed in Figure 4, the first reaction rule reacts to the change of the belief count, which is about the current value of the corresponding observable property in the artifact myCounterTool, declared to be used in plan by means of the using: attribute (myCounterTool in this case is known because it is a parameter of the task MyTask that triggered the plan). The first case includes also changes to beliefs that used to keep track of the execution state of those actions that have been explicitly labelled. In this case the event is syntactically represented by when done_*AL* or when failed_*AL*, where *AL* is the action label. In Figure 4 the second rule says that the inc operation is performed on the myCounterTool when the action someAction1 completed.

The second case concerns changes to beliefs that represent information that can sent by other agents through the tell communicative action. In that case the syntax is told *What* [by Whom], where *What* and Whom are respectively beliefs declared by the agents about the type of expected information and – if specified – the role of the agent sending the message. In Figure 4 the third rule says that the agent prints a message on the *console* artifact when the agent receives a message from an agent of type RoleB about the ping belief.

Finally, the third case concerns changes to the internal belief (called time), which keeps track of time passing, in milliseconds. The fourth rule in Figure 4 says that a meetingRequest information is told to a friend agent at the time specified by the alarm local belief.

Besides these three cases, some predefined keywords are used to identify directly some specific events: among the others, always is used to mean that the rule is always triggered—the event in this case is *at each new cycle*, where the agent cycle is explained in Section 3.7, or, equivalently, the belief about the value of the agent inner logical clock has been incremented. Two rules in Figure 4 use always, one including also the context so that the action in that case is executed every time the info is less that one hundred.

3.6.2 Actions

Actions specified in action rules can be either *external* actions, that are those affecting the environment or communicative actions to send messages to other agents, or *internal* actions, affecting just the internal state of the agent. The actions on the environment are actually given by the operations provided by artifacts—in this case the name of the operation is specified, along with parameters value and (optionally, when needed) the target artifact. Examples are shown In Figure 4 in the second and third rules. Parameters are nominal/keyword-based, so the name of the parameter plus the value must be specified. Some predefined external actions are provided to dynamically create and dispose artifacts, to spawn and terminate agents, to communicate with other agents, and other related functionalities Communicative actions currently include tell, specifying the belief to tell and the receiver of the message (see the fourth rule in the example), and the dotask, to assign a task to an existing agent.

Basic internal actions include the assignment action, to update the value of a belief, and actions to work and manipulate local (Java) objects, to instantiate local objects and to invoke methods—adopting a Java-like syntax. Also control structures such as if, while and for are provided as specific internal actions. A particular important and useful internal action is the one that allows for instantiating anonymous *sub-plans*, as a way to define blocks of action rules. An anonymous sub-plan is like a plan but without the specification of a name and of the task attribute. Syntactically, they are represented by { ... } blocks. An example is given by the last rule in Figure 4. In that case, the action which is performed when the count belief is updated and the value is less than zero is the instantiation and execution of a sub-plan, composed by some other action rules. From the execution point of view, sub-plans are stacked upon their parent plan—similarly to what happens with procedure calls.

```
plan SeqPlan1 task: ... goal: done_a3 {
  todo_a1 => action1 #a1
  when done_a1 => action2 #a2
  when done_a2 => action3 #a3
}

plan SeqPlan2 task: ... {
  action1
  action2
  action3
}

plan ParPlan task: ... {
  action1 |
  action2 |
  action3
}

plan ParPlan2 ... {
  action1 #a1 |
  action2 #a2
  done_a1 && done_a2 => action3
}
```

Figure 5. Plans with sequences of actions and workflows of actions.

3.6.3 Specifying Mixed Active and Re-Active Behaviour

The combination of the event and condition attributes make it possible to write down active behaviors exhibiting the desired level of reactivity, from simple sequences of actions to purely reactive behaviors to workflows of actions mixing the styles.

The execution of a sequence of actions action1, action2, action3, ... can be encoded in terms of actions rules as shown in the SeqPlan1 plan in Figure 5: in that case, the action action2 is executed when the previous one has completed (event when done_a1), and the action action1 is executed only if it has not been already executed once. done_*XX* and todo_*XX* are beliefs storing boolean values, that are automatically added to the beliefs when the plan is instantiated (as part of plan's beliefs), initially set respectively to false and true. Under the hood, when an event concerning the success of an action labelled *AL* is perceived, then the belief done_*AL* is updated to true, and internally when an action labelled with *AL* is executed, the belief todo_*AL* is set to false.

To easy the specification of sequential behavior, some syntactic sugar and convention have been introduced. When no event and context is specified – and so only the action is – the action can be selected to be executed immediately and it is executed only once. Also, the agent must wait the completion of the action before eventually selecting the next action, in the case that also the next action rule is without the specification of the event and the context. Given this

```
plan MixedPlan task: MyTask, using: myCounterTool {
  c: int = 0;

  inc (delta: 1)
  inc (delta: 5)
  inc (delta: 2)
  +count => atomically: {
    println (msg: "new value: "+count)
    c = c + 1
  }
}

plan AnotherPlan task: ...
            using: console @ main, counter @ wsp2 {
  max : int = 100;

  println (msg: "starting")
  goal: count > max {
    always => {
      inc (delta: 1)
      println (msg: "incremented ")
    }
  }
  println (msg: "done.")
}
```

Figure 6. Plans integrating sequences of actions and reactions.

convention, the sequence can be expressed simply by a list of actions, as shown in the SeqPlan2 plan.

A further convention is adopted to allow for easily specifying sequence of actions in which we don't want to wait for the completion of each action before executing the next one. This can be done by adding a pipe character — between two actions – that is, after the action expressed in an action rule, which can have also the event and the condition specified, and before an action rule with only the action specified. An example is given by the ParPlan plan, where the actions are selected and executed sequentially, however without waiting the completion of any of them, that is action2 is executed without waiting for the completion of action1 and action3 is executed without waiting for the completion of action2. A further example is given by the ParPlan2 plan, where action2 is executed without waiting the completion of action1. Instead, action3 is executed only when both action1 and action2 have completed (which is a context condition, without events).

This model allows for specifying quite naturally plans that combine sequences, reactions, workflows—obtaining behaviors in which the agent is actively doing some sequence of actions and at the same time is able to react to events, eventually suspending what is doing in order to do some other action. A simple example of this behaviour is shown in Figure 5, where myCounterTool artifact is used by the agent executing three times the inc action (operation provided by the artifact), and each time the count observable property changes, the agent reacts by printing the actual value on standard output and incrementing a local belief. A final example is given by AnotherPlan, in which the agent first prints a message on the console, then increments a counter artifact until the perceived value of the observable property is greater than a max value.

3.7 The Agent Execution Cycle

At the architectural level, the integration of a task-oriented and event-driven behavior is made it possible by the *agent execution cycle* – also called *control loop* – which defines the conceptual sequence of steps that the agent processor (interpreter) repeatedly executes to run an agent, until agent termination (if the agent terminates). Conceptually the agent processor can be conceived as a machine with a clock, executing an execution cycle at each clock tick. The model of agent execution cycle adopted in simpAL is a simplification of the one typically adopted in BDI architecture, in particular of the one found in AgentSpeak(L) [13]/Jason [5] programming languages. It is composed by three conceptually different stages (see Figure 7), that are repeatedly executed: *sense*, *plan*, and *act*.

Sense stage – In this stage the internal state of the agent is updated by processing those *external* events coming from the environment and stored in an agent external event queue. Such events may concern: *(i)* changes of the observable state of the environment – more specifically, the set of artifacts – that are relevant for the agent work and that the agent is using; *(ii)* the completion with success or failure of an environment action – corresponding to the execution of an artifact operation – that the agent executed previously; *(iii)* the arrival of messages communicated by other agents.

The processing of every external event which is relevant of the agent causes the update of the belief base. In particular, changes to the observable properties of an artifact cause the update of the related beliefs in those plans in which the agent declared to use the artifact. The arrival of a message related to communication acts informing the agent about some information causes the update of the local beliefs of the plans in execution for which the such information is relevant. Finally, external events related to action success or failure cause the updating of the beliefs as described in previous section.

Besides the informational state, also the motivational state and the intentional state may be updated in this stage. If the agent receives a message related to a *do-task* communicative act, then a new task to do is added in the agent *task to-do list*. The same applies if the agent has some perception from the environment that there is a new task to do. The intentional state is updated instead when receiving an event about an action success or failure, in particular the state of the plan in execution that selected the action is updated accordingly.

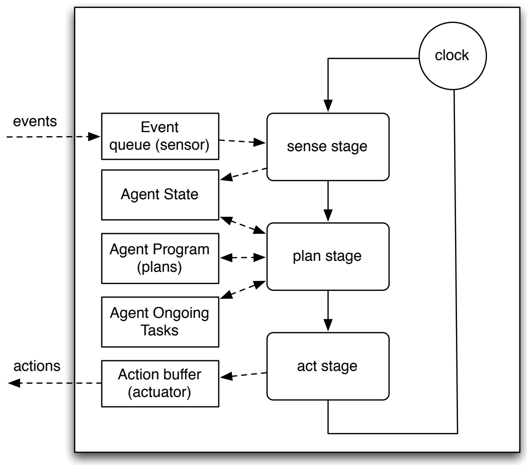

Figure 7. Conceptual representation of an agent architecture, with in evidence the stages of the execution cycle.

For each update to the agent beliefs caused by external events an *internal* event is generated and added into an internal event queue, which will be accessed in the *plan* stage.

Plan stage – In this stage, the next action to do is selected, given the content of the internal even queue, the current state of the agent, and the behavior specified by plans of the agent script. First, the ongoing task (and plan in execution) to focus on is selected according to a basic scheduling policy. This because an agent may perform multiple tasks concurrently: for each ongoing tasks there is a stack of plans in execution. It is a stack because the execution of a plan may involve the execution of sub-plans and/or sub-tasks, which cause the instantiation of plans that must be completed before the "parent" plan would proceed. These sub-plans are then pushed on top of the parent plan, in the plan stack related to the same task. Once chosen the ongoing task to focus, the action selection is driven by the set of action rules currently available given the stack of plans related to the task.

Act stage – In this stage the action selected in the *plan* stage is executed. If the action is internal, the effect is just an update of the internal state of the agent; if the action is on the environment, the effect is to trigger the execution of a corresponding operation on some artifact (this will be described in Section 4). The completion of the action or its failure will be possibly perceived as an asynchronous event in one next future cycle.

Conceptually, the agent processor continuously executes these three stages, performing one execution cycle at each logical clock tick. Conceptually, the agent control flow is never blocked: so, for instance, even if the agent has executed some kind of blocking action – that means in our case an action in which the completion event will be perceived not immediately but some time in the future – it can always react if the current plan prescribes this.

```
artifactmodel Counter {
  obsprop count: int;
  operation inc(delta: int);
}

artifact MyCounter implements Counter {
  init(startValue: int) {
    count = startValue;
  }
  operation inc(delta: int) {
    count+=delta;
  }
}
```

Figure 8. Example of a model of artifact (top) and of a template implementing the model (bottom).

4. The Environment Abstraction

The environment of an organization is given by the overall dynamic set of artifacts, distributed in workspaces, part of which are available by default to provide basic functionalities. For instance, in each workspace there is a *console* artifact providing a println operation (action) to write on the console. Basic functionalities include the creation or disposal of artifacts, lookup for existing artifacts satisfying some criteria, and many other management aspects that are not discussed in this paper – including security.

As mentioned in previous sections, an artifact can be conceived as a module encapsulating the implementation and execution of sets of operations as actions that the artifact makes it available to agents, and a set of observable properties that agents using the artifact may perceive. Besides, an artifact contains also some hidden (not observable) state variables, useful for implementing artifact functionalities, and possibly also internal operations, which can be executed by other operations of the artifact and are not part of the usage interface.

Analogously to the agent case, we separate the abstract description of the artifact functionalities from their concrete implementation defining artifact structure behavior. The former is specified in *artifact models*, defining the usage interface of all the artifacts implementing that model. The definition of a usage interface includes a name of the interface, a set of observable properties and the declaration of a set of operations. Figure 8 (top) shows a very simple example of model of a counter artifact, providing a count observable property and a inc operation. Observable properties are similar to variables, characterized by a name, a value and a type. The parameters declared by operations are keyword based—for instance, inc has a parameter called delta.

The artifact implementation is defined in *artifact templates*. Like classes in OOP, artifact templates are a blueprint for creating instances of artifacts. The definition of an artifact template includes a name, the declaration of the implemented artifact model, the concrete implementation of operations – including internal operations – and the definition of those internal variables that are used in operation implementations. Figure 8 (bottom) shows an example of template, implementing the Counter model.

Operation behavior is given by a simple sequence of statements, in pure imperative style, using classic control flow constructs, assignment operators, etc. As mentioned previously Java is used as a language for defining data structures. So objects as well as primitive values can be used in expressions and as value of variables and observable properties, and method invocation appears among the statement of the language.

A main aspect of the model is that operation execution is *transactional*, in the sense that *(i)* changes to the observable state of the artifact (properties) are done atomically, *(ii)* changes are perceived by agents observing the artifact only when the operation completes (with success). The execution of an operation can fail: this causes the failure of the action on the agent side, on on the artifact side the observable state is rolled back to the value before executing the operation.

4.1 Typing Artifacts

Artifact models are used to define the type of artifacts. This makes it possible to: *(i)* check errors in agent scripts—each time an artifact identifier is used either to define explicitly the target of action execution or the source of a belief related to an observable property, we can check that the specified action or observable property is actually defined for that type of artifact; *(ii)* check errors in artifact templates—by checking that every operation declared in the model is implemented then in the template; *(iii)* defining a sub-type relationship so as to enjoy substitution property for artifacts. That is: if an agent needs to use an artifact implementing a model M, then it can actually use any artifact implementing a model M' which is a subtype of M. *(iv)* exploiting polymorphism also at the artifact level. That is: two artifact (templates) implementing the same model M can provide different implementation and finally behaviors for the same operation (action). So, on the agent side, different kind of artifacts implementing the same model are used in the same way, without the need of knowing the specific implementation of the artifact, yet possibly obtaining different specialized behaviors depending on the specific template.

Finally, implementation reuse and incremental development is enabled by exploiting *inheritance* among artifact templates. In particular, an artifact template can be defined by extending/refining an existing artifact template: in that case it inherits the implementation of operations, observable properties, state variables, which can be extended by defining new operations or refining existing ones.

```
orgmodel MyOrgModel {
  workspace w1 {
    console: Console;
    c1: Counter;
  }
  workspace w2 {
    console: Console;
    bb: Blackboard;
  }
  agents {
    agent0: MyRoleA;
    agent1: MyRoleB;
  }
}

agentscript MyScript implements MyRoleA
                     in MyOrgModel {
  plan MyPlan... using: bb @ w2 {
    putNote note: new Note("hello").
  }
}

org MyOrg implements MyOrgModel {
  workspace w1 {
    c1 Counter startValue: 10
    ...
  }
  ...
  agents {
    agent0 script: MyScript task: MyTask(aParam: 10);
    ...
  }
}
```

Figure 9. Definition of an organization model, of an agent script specifying also the organizational model, and of an organization implementing the model.

5. The Organization Abstraction

The "main" of a simpAL program corresponds to definition of the concrete (initial) configuration of an organization. Also for this aspect we separate the model part from the concrete implementation one. The *organization model* defines abstractly the structure of the overall program, which includes the set of workspaces and possibly the name (identifier) and type of some agents and artifacts that are known to be part of the organization.

Figure 9 shows a simple example: MyOrgModel defines a model of organization composed by two workspaces called w1 and w2. The workspace w1 hosts a console artifact of type Console and a c1 artifact of type Counter. A console artifact is declared also in w2, along with a Blackboard artifact. In this case c1, console, bb are the names (identifiers) of the artifacts—actually the full identifier is given by the name and the name of the workspace (e.g., bb @ w2), since artifacts with the same name can be run in different workspaces. Then, inside the organization agent0 and agent1 are declared as the identifiers of two agents playing the roles MyRoleA and MyRoleB. It is worth remarking that the declared agent/artifact instances are not meant to be the unique instances of agents and artifacts inside the program: only those whose name and type must known at the organizational level. The declarations are useful to define global symbols that can be resolved and checked in scripts that explicitly declared to play inside an organization of this type (see Figure 9). By doing so, the symbols and identifiers declared in the organization model can be referred also in the script (e.g., bb @ w2 in using: attribute) and then checked at compile time.

Then, the definition of a concrete organization accounts for specifying the concrete instances of agents and artifacts declared in the org model (see Figure 9). For artifacts, the artifact template is specified, possibly including also the value of some initialization parameter. For agents, the script must be specified, along with the initial task to do and task initial parameters. Actually, it is not necessary to specify the instantiation of all the agents and artifacts declare in the org model: some can be also instantiated dynamically.

The definition of the organization can include also information about its distributed deployment (not discussed here for lack of space), so specifying the *organizational nodes* hosting the execution of workspaces and of agents. Organizational nodes are bound to physical network nodes when launching the program.

6. The simpAL Platform: A Sketch

The simpAL platform is being developed in Java and is available as an open-source project at `http://simpal.sourceforge.net`. The platform includes:

- an Eclipse-based IDE organized in plug-ins, currently including an xtext-based editor;
- a *compiler*, to compile simpAL programs into executables that can be run by a *launcher*, which loads the program on the runtime infrastructure to execute it;
- a distributed *runtime infrastructure* (called simpAL OS), which must be in execution in every node / host which may need to execute (a part of) a simpAL organization;
- libraries, including a *system library* – with artifacts providing basic system functionalities related to I/O, GUI, file system access – and a *utility library* – with artifacts providing coordination functionalities (e.g. blackboards, tuple spaces, etc).

Besides the editor, an important part of the future development of the IDE concerns the development of: *(i)* an interactive programming environment, to visualize and interact with simpAL programs while developing them; *(ii)* a debugger, and a *(iii)* profiler. These front-end tools will be necessarily integrated with proper backends, part of the runtime infrastructure.

7. Related Work

The work presented in this paper is strongly related to existing agent programming languages introduced in the (D)AI literature, in particular to those based on the BDI model—interested readers can refer to [4, 6, 7] as comprehensive surveys. As mentioned in the introduction, differently from these languages simpAL has not been designed with AI and DAI problems in mind, but as a language to explore agent-oriented programming as a mainstream programming paradigm for software development. For this reason, on the one hand simpAL has features that are typically not interesting from a AI point of view – such as a well-defined notion of type for agents, mechanisms for incremental development and reuse, etc.; on the other hand, it does not have features that are important when an AI context is chosen instead, such as using first-order logic to represent beliefs, goals and related inference rules.

Besides existing agent programming languages in (D)AI context, our work is related to existing frameworks and platforms that allow for developing agent-oriented programs exploiting existing programming languages. A main example is JADE [3], one of the most used Java-based FIPA compliant platform for developing agent-based software. JADE makes it possible to write agent programs in Java, where agents communicate using FIPA ACL as a standard high-level agent communication language. The model adopted for defining agent is based on *behaviors*, which share some similarities with the notion of plan adopted in simpAL. Besides agents, recently JADE model has been extended with a notion of *service* which is quite similar to the notion of artifact used in simpAL and in A&A, as non-autonomous components providing operations to agents.

In the context of concurrent programming, the agent-oriented abstractions in simpAL can be seen as a high-level extension of the concept of actors, active objects, processes and similar concepts – introducing specific first-class concepts to improve the structuring of autonomous behaviors (tasks, plans), the integration of task-oriented and event-driven behaviors, the separation of concerns related to autonomous (agents) and non-autonomous (environment) entities.

Actually, in concurrent programming the notions of actor and agent are often used as synonyms, to generically refer to active entities that exchange asynchronously messages and are reactive, in particular react to messages received in input by other agents. An example is given by recent works exploiting the F# functional language – and its asynchronous programming model – to implement agent-based concurrency and agent-oriented programs on top of functional programs [19]. Always in the context of concurrent programming, a notion of agent is used also in the Clojure language [10], as a state-full, reactive, non-autonomous entity which is used as a simple concurrency mechanism to manage the execution of asynchronous I/O operations. A common issue which is considered very important by these approaches as well as in simpAL is the the availability of some programming support for easily integrating active and reactive behavior, to build software components that are able to react to events coming from the environment (which in the case of actors are given by messages sent by the other actors) while they are running. This is a central point tackled also in the implementation of actors as a library on top the Scala language [8].

Finally, simpAL is related to our previous work simpA [16], a Java-based framework for developing concurrent concurrent programs using agent-oriented abstractions. simpAL can be considered an evolution of that work, *(i)* introducing a new language instead of relying on a framework based on existing technologies – this is essential to investigate aspects such as typing, and *(ii)* adopting a different model for defining agents, based on BDI, not exploited in simpA.

8. Conclusion and Future Work

The investigation of agent-oriented programming as general-purpose programming paradigm calls for the development of languages and platforms that make it possible to exploit the level of abstractions, concepts and features that are already found in many existing agent programming languages, but extending them so as to be effective for the theory and practice of programming. This is the objective of the simpAL project, and in this paper we presented a first frame about the essential concepts that characterize the language, along with some sketch about the development of its platform and infrastructure.

simpAL is not meant to be conceived as the umpteenth agent-oriented programming language introduced in DAI context or as an extension or improvement over existing languages in that context. Instead, it is meant to be a first comprehensive approach to explore in theory and practice agent-orientation as a mainstream paradigm for programming and developing software systems, as an evolution of the object-oriented one.

The finalization and improvement of the language and of the platform is a main part of our current and future work, along with a formalization of the language and a concrete investigation of its effectiveness and limitations by developing non-toy programs.

References

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

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

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

[4] R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming Languages, Platforms and Applications - Volume 1*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, 2005. Springer.

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

[6] R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming Languages, Platforms and Applications - Volume 2*, Multiagent Systems, Artificial Societies, and Simulated Organizations, 2009. Springer.

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

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

[9] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[10] R. Hickey. Agents and asynchronous actions (in clojure), 2011. Online document, available at: `http://clojure.org/agents` – Last Retrieved: Sept. 1, 2011.

[11] H. Lieberman. The continuing quest for abstraction. In *ECOOP 2006 ? Object-Oriented Programming*, volume 4067/2006, pages 192–197. Springer Berlin / Heidelberg, 2006.

[12] 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.

[13] A. S. Rao. AgentSpeak(L): Bdi agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world (MAAMAW'96)*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

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

[15] 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.

[16] A. Ricci, M. Viroli, and G. Piancastelli. simpA: An agent-oriented approach for programming concurrent applications on top of java. *Sci. Comput. Program.*, 76:37–62, January 2011.

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

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

[19] D. Syme. Async and parallel design patterns in f#: Agents, 2010. `http://blogs.msdn.com/b/dsyme/archive/2010/02/15/async-and-parallel-design-patterns-in-f-part-3-agents.aspx`, Last Retrieved: Sept 1, 2011.