Programming Distributed Multi-Agent Systems in simpAL

Andrea Santi
DEIS, University of Bologna
Cesena, Italy
Email: a.santi@unibo.it

Alessandro Ricci
DEIS, University of Bologna
Cesena, Italy
Email: a.ricci@unibo.it

***Abstract*—Distribution is one of the essential features characterizing multi-agent systems (MASs), giving developers the opportunity to seamlessly conceive and then engineer a physically distributed application as a MAS spread among different network nodes. Nevertheless, the current support given by state-of-the-art Agent Programming Languages (APLs) and related platforms for programming distributed multi-agent systems as well as for handling distribution when deploying, running, debugging the distributed MAS is still quite primitive. In this paper we tackle this problem by introducing simpAL, a new agent-oriented programming language and platform which has been conceived from the beginning to provide a more comprehensive support for programming, deploying and executing physically distributed MASs.**

I. Introduction

Multi-Agent Systems (MASs) are a main paradigm for designing and developing complex software systems [12], *distributed* systems in particular. Agent programming languages (APLs) have been introduced to ease the development of MASs, providing agent-oriented first-class abstractions directly at the language level [4], [5], [6]. To deal with distribution – i.e., programming *physically* distributed MASs – the support provided by state-of-the-art APLs is typically limited to mechanisms enabling the communication among agents and by services available at runtime that provide facilities such as agent discovery, message routing, ontology management, and so on. The program of a distributed MAS is typically given then by a set of *separate* MASs programs, to be deployed, run and managed by hand or by using OS scripts on the distinct network nodes. This makes the programming as well as the deployment and execution of physically distributed MASs using state-of-the-art APLs significantly harder than the centralized case, i.e. MASs running on the same host.

In this paper we aim at tackling this problem and, finally, easing the programming of physically distributed MASs, by devising at the language level proper abstractions and mechanisms suitably supported by the runtime. To this purpose, we will use a novel agent-oriented programming language called simpAL. simpAL has been devised with the purpose of exploring agent-oriented programming as a high-level general purpose programming paradigm for concurrent and distributed programming, as a natural evolution of object-oriented and actor-based approaches [17]. The language is based on the A&A (Agents and Artifacts) meta-model [14], thus providing first-class abstractions to define agents – based on a BDI-like model [16] – and the distributed environment where the agents are logically situated, in terms of artifacts and workspaces. Concerning distributed programming in particular, simpAL provides specific language abstractions such as the notion of *organization* and *workspace* and related runtime support that ease the programming as well as the deployment and execution of distributed MASs.

The remainder of the paper is organized as follows: in Section II we provide the background of this contribution by relating the current support of state-of-the-art APLs for realizing physically distributed MASs with the one we want to obtain in simpAL. In Section III we present the main concepts of the language, and then we describe how a distributed multi-agent system can be programmed in simpAL (Section IV) and executed on the simpAL distributed runtime infrastructure (Section V). Finally, we conclude the paper by highlighting the main good points of the approach (Section VI) as well as its current limitations and related ongoing and future work (Section VII).

II. Background

Many APLs have been proposed so far in literature for the programming of agents and multi-agent systems [4], [5], [6]. Recently, a strong effort has been put in devising *practical* languages, i.e. thought to be used primarily for concretely programming MASs and not (only) for theoretical investigations. Main examples include Jason [3], GOAL [10], 2APL [8], AFAPL [19], JACK [11].

Such languages and platforms provide a good support for developing agent and multi-agent programs running on a single host; conversely, the support provided for the engineering, deployment, execution and management of *physically* distributed MASs – i.e., MASs running on multiple network nodes – is still quite primitive [7]. It is essentially limited to:

- enabling direct communication among agents based on some agent communication languages – such as FIPA ACL [1] or KQML [9] – or indirect ones, based on some coordination abstractions or the environment [15];
- exploiting some well-known *services*, typically represented by facilitator agents, provided by default by the runtime infrastructure to help agent discovery, message routing, interoperability, ontology management, etc.

```
/* .mas2g file related to the MAS1  */
agentfiles {
  "agent0.goal".
}
launchpolicy {
    launch a0:agent0.
}
```

```
/* agent a0 */
init module {
  beliefs{
    started(true).
  }
  ...
}
main module {
  if bel(started(true)) then sendonce (a1, helloMsg)
    + insert (sentMsgToA1).
  if bel(sentMsgToA1) then sendonce (ag2, hello_msg)
    + insert (sentFirstMsgToA2).
  if bel(sentFirstMsgToA2) then sendonce (a2, !do_job).
  ...
}
```

```
/* .mas2g file related to MAS2 */
agentfiles {
  "agent1.goal".
  "agent2.goal".
}
launchpolicy {
  launch a1:agent1.
  launch a2:agent2.
}
```

```
/* agent a1 */
...
event module {
  forall bel(received(Sender, hello_msg))
    then insert(helloMsgReceived(Sender)).
}
```

```
/* agent a2 */
...
event module {
  forall bel(received(Sender, imp(doJob)))
    then insert(newDoJobGoalDelegated(Sender)).
}
```

Fig. 1. An example of physically distributed MAS programmed in GOAL highlighting: *(i)* the need to define a separate MAS (in this case two) for each node in which the distributed application needs to execute, and *(ii)* typical programming errors related to message-based interactions that can be detected only at runtime (i.e., the sending of wrong messages to agents (bottom-left line 9), the use of misspelled agents identifiers (bottom-left line 11), the assignment of wrong goals to agents (bottom-left line 13)).

To this end, almost all APLs in the state-of-the-art are provided with a FIPA-compliant platform, often based on existing middleware such as JADE [2]. Given this support, programming a physically distributed MAS on these APLs concretely means creating a collection of separate programs, each one meant to be run on a different host. Each program will spawn then agents that eventually will locate themselves at runtime by exploiting discovery services and will communicate by exchanging ACL messages. A concrete example written in GOAL – other APLs could have been chosen as well since the support they provide to program a physically distributed MAS is analogous – is reported in Fig. 1, in which a physically distributed MAS is programmed defining two separated MASs (i.e., `MAS1` and `MAS2`). In the example agent `a0`, part of `MAS1`, interacts via speech acts with both agent `a1` and agent `a2`, part of `MAS2`, which is in execution in a separated network node.

On the one side, this approach is quite effective to handle dynamism and openness giving hence MASs the opportunity to evolve freely, on the basis of the interaction dynamics experienced at runtime. For example: *(i)* new parts can be dynamically added to running applications by simply launching new MASs and then making them interact with existing ones, *(ii)* new agents, possibly unknown at design time, can be instantiated, made interact with MASs already in execution and also dynamically looked up by the original agents of the MASs through the exploitation of discovery services.

On the other side, this approach makes the programming as well as the deployment and execution management of distributed MASs quite complicated and troublesome jobs [7]. From the programming point of view, a main problem is the lack of proper abstractions and mechanisms that allow for discovering *errors* related to message-based interactions among agents *before running* the MAS. In this case the problem is not related to *physical* distribution, but simply *distribution* (of control) among multiple agents that need to interact. An example of the problem is shown in Fig. 1 using GOAL. In GOAL all the errors related to message-based interactions such as the sending of wrong messages to agents (Fig. 1 bottom-left line 9 where `helloMsg` is used instead of `hello_msg`), the use of misspelled agents identifiers (Fig. 1 bottom-left line 11 where `ag2` is used instead of `a2`), the assignment of wrong goals to agents (Fig. 1 bottom-left line 13 where `do_job` is used instead of `doJob`), etc. can be only detected at runtime.

Discovering these kinds of errors only by running the MASs makes the development hard and time-consuming, especially when the MASs are physically distributed. To this purpose, we aim at having suitable abstractions at the language level that would allow to detect such programming errors statically and systematically, in order to: *(i)* reduce the cost of errors detection from both a temporal and economic point of view, and *(ii)* avoid complicated – and possibly long – debugging sessions for detecting errors at run-time—e.g., an error that occurs only after several complex computations and long interaction dynamics.

From the point of view of the management of MASs deployment and execution, we want the runtime to hide all the complexities related to the physical distribution, so that physically distributed MASs are launched, terminated and managed in the same way of MASs that are running on a single node. This is not possible in current APL, where this kind of management still need to be done by hand. In fact, in order to deploy and run a physically distributed MAS, one has typically to deploy by hand on the different nodes the various parts that constitute the MAS, and then execute the whole program by launching in proper order the different parts—i.e., taking into account application-specific constraints in order to

guarantee a correct initialization of the whole system. To stop the MAS, one needs typically to terminate by hand all the individual parts.

For instance, taking Jason to make a concrete example – but as well as in GOAL, AFAPL, etc. – the runtime execution of a whole physically distributed MAS is managed exploiting a FIPA-compliant agent platform – JADE in this case – in order to create a distributed runtime infrastructure among the interested network nodes. To this end, a typical execution of a distributed MAS in Jason follows this dynamics. First, the program is manually deployed in the target network nodes. Terminated the deploy process, the first part of the distributed MAS to be executed is launched, manually, and the related Jason runtime takes in charge the creation of a main JADE container for the distributed application. Then, all the other parts are launched in the desired order, manually one by one, and for each of them the dedicated Jason runtime creates under the hood a new JADE container linked to the main one, in order to create the distributed runtime platform that has in charge the execution of the whole program. Also application termination needs to be managed by hand, manually shutting down on each node the different Jason applications.

Dealing with these processes by hand is both tiresome – i.e., the steps described above need to be repeated at each launch – and error-prone – e.g., miss the deployment of updated sources in some node, wrong initialization sequence of the different application parts, etc. – thus calling for a better, possibly automatized, support for their management. We argue that this goal can be made easier by having specific first-class abstractions/constructs at the language level that make it possible describe a MAS along with its (possibly) physically distributed structure. The introduction of such constructs can then make APLs' runtime infrastructures aware of all those information required to manage the deployment and the coordinated launch/termination of physically distributed MASs in a automatized manner, limiting as much as possible developers intervention, since the information needed to manage such processes has become part of the program itself.

In the following, we discuss how all the issues concerning the programming and execution management of physically distributed MASs introduced in this section are managed by simpAL both at the programming and runtime/infrastructure level.

III. Programming Model Overview

The programming model adopted in simpAL for designing and implementing MASs integrates concepts defined in the A&A (Agents and Artifacts) conceptual model [14] and the BDI (Belief-Desire-Intention) agent model [16]. A distributed MAS is designed and programmed as an *organization* of agents working together inside a shared possibly distributed *environment* organized in *workspaces*.

Agents in simpAL are task-oriented entities designed to perform autonomously some *tasks*, possibly interacting with other agents via message-passing and with the environment where they are situated. They pro-actively decide what are the best actions to perform and when to do them, reacting to relevant events from their environment, fully encapsulating the control of their behavior.

The environment plays a key role in mediating and supporting members' individual and cooperative tasks. In simpAL – as well as in A&A – the environment is modularized into a dynamic set of first-class computational abstractions called *artifacts*, which represent the resources and tools that agents share and may exploit concurrently and cooperatively. Artifacts are useful to directly model non-autonomous software components, encapsulating and modularizing functionalities that can be suitably exploited by agents. Examples of artifacts are bounded buffers, a clock, a database, an external web service, etc. Like artifacts in the human case, artifacts in simpAL can be dynamically instantiated and disposed (by agents), and, when needed, designed to be composed so as to create complex artifacts by connecting simpler ones.

Then, a further concept is needed in order to explicitly define the overall structure of the program which can be physically distributed over (possibly) different network nodes. To this end, in simpAL we introduce the notion of *workspace*. The overall set of agents and artifacts of an organization may be partitioned into a set of workspaces as logical containers, possibly running on different nodes of the network, defining the logical structure of the application. Actually, while being located in a specific workspace, an agent can work concurrently and transparently also with agents and artifacts of other workspaces belonging to its organization.

Finally, pure objects – as defined in modern OOP – are used to define the data model (and related purely transformational computations, e.g. retrieving the value of an object field) of programs. That is, agents, artifacts and workspaces are meant to be used as coarse grained abstractions to define the shape of the organization (i.e., of the program), in particular of the control part of it (decentralized, distributed). Objects are then the basic data structures used inside agents, artifacts and related communications and interactions.

IV. Programming Distributed MASs in simpAL

In simpAL, a physically distributed multi-agent system is programmed as an *organization* whose logical structure is defined in terms of workspaces distributed among different network nodes. In this perspective, the notion of workspace represents the key to conceive and model the logical structure of the MAS, by properly grouping, on the basis of both application requirements and decisions made at design time, agents and artifacts in the set of workspaces that define the organization. This grouping is meant to define only the logical structure of the application – i.e. the organization – abstracting from all the details concerning its deployment, which is instead managed through the definition of a run/deploy configuration, specified into a dedicated configuration file (see the next subsection).

In this section we show how a distributed multi-agent system can be programmed in simpAL using a guiding toy

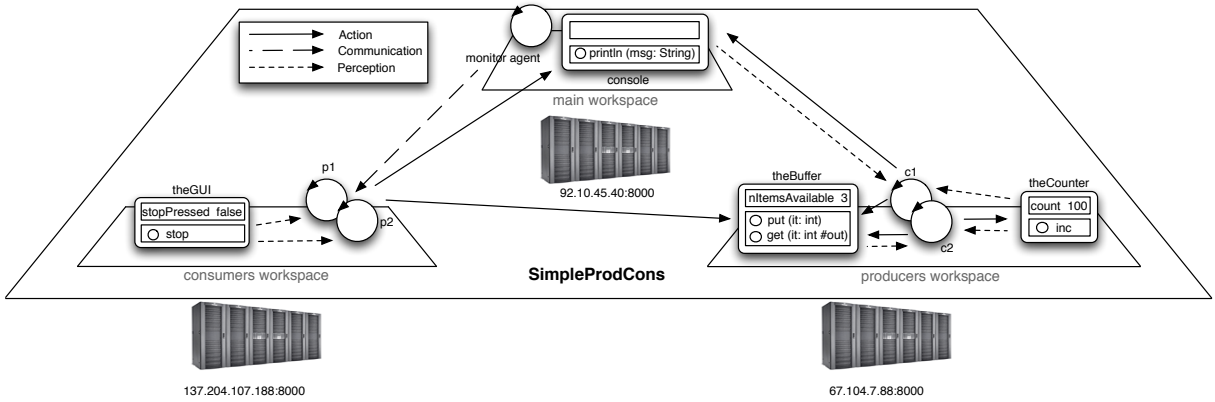

Fig. 2. An abstract view of the producers-consumers organization, distributed among three different simpAL nodes.

example, based on a slightly revised producer-consumer architecture (see Fig. 2). The source code of the example is available for download on the simpAL website, as part of its standard distribution. In the example, a set of producer agents have the task of producing continuously some items that must be consumed by a bunch of consumer agents. Consumer agents must stop their activities as soon as the total number of items processed is greater than a certain value. Also, producer agents must stop the production as soon as the user stops it through a GUI. A monitor agent – observing the overall activities – may communicate directly to the producer agents that more items need to be produced. Some artifacts are exploited to support agent work and coordination: a bounded buffer artifact (with the obvious functionalities), a counter (used by consumers to keep track of the overall number of items processed) and a GUI (used by producers to observe user inputs).

In the following we proceed top-down. First we focus on the programming of the organization, defining the distributed structure of the system, and then we introduce the programming of agents and artifacts as basic components of a simpAL program.

A. Defining the Organization

Following the basic principle of separation between interface and implementation, the definition of a simpAL organization is characterized on the one side by the notion of *organization model*, which is introduced for specifying the description of the abstract structure of the overall program, and on the other side by the notion of a concrete *organization* which allows instead to define a concrete application instance referring to an existing *organization model*. An organization model is identified by a proper name and it defines:

- The workspace-based logic structure of the organization, in which, for each workspace, it is possible to define, statically, the name (identifier) and the type of agents and artifacts that, for that workspace, will be automatically instantiated and initialized at each launch of the organization. Dynamic instantiation is addressed later on.
- The set of roles and the set of artifact models defining the types that can be used, for agents and artifacts, in the context of that organization.

Fig. 3 shows a simple example of an organization model definition that refers to our producer-consumer sample. As a design choice we decided to structure the topology of our application in three workspaces (but other topologies could have been chosen as well): `producers`, `consumers`, and `main`—the latter is available by default in every organization. The workspace `producers` hosts a couple of producer agents (`p1,p2`) and the GUI artifact (`theGUI`), while the workspace `consumers` hosts a couple of consumer agents (`c1,c2`), the counter and the buffer artifacts (`theCounter`, `theBuffer`). Finally, the `main` workspace hosts the monitor agent (`monitor`) and a `console` artifact used for purely logging functionalities.

An organization can contain further agents/artifacts instances besides those statically declared in the organization model—since both agents and artifacts can be dynamically spawned and created by agents by means of specific actions. The static case is useful anyway to specify the identifier of those components whose name and type must be known at the organizational level, in other words to define global symbols that can be resolved, in a transparent manner w.r.t. where they will be actually deployed, and checked in that simpAL sources *explicitly declared* in the context of an organization of this type (see next subsection).

Then, the definition of a concrete organization accounts for specifying the concrete instances of agents and artifacts declared in the organization model. For artifacts, the artifact template – i.e., the name of the source containing the artifact implementation, see Section IV-C – is provided, possibly including also the value of some parameters required by the artifact initialization operation. For agents, the initial script (see Section IV-B) to be loaded must be specified, possibly including parameters needed to boot the script, along with the initial task to do, and task initial parameters. Fig. 5 shows an example of a `SimpleProdCons` organization for the case of our producer-consumer sample, where the agents/artifacts declared into the `ProdConsOrg` organization model are here properly initialized in order to characterize the specific `SimpleProdCons` organization instance.

```
/* ProdConsOrg organization model */
org-model ProdConsOrg {

  roles: Producer, Consumer, ProductionMonitor;
  interfaces: Console, GUI, Buffer, Counter;

  workspace producers {
    p1, p2: Producer
    theGUI: GUI
  }

  workspace consumers {
    c1, c2: Consumer
    theCounter: Counter
    theBuffer: Buffer
  }

  workspace main {
    console: Console
    monitor: ProductionMonitor
  }
}
```

Fig. 3. Example of a simple organization model.

```
/* SimpleProdCons organization deployment file */
workspace-addresses {
 main = localhost
 producers = localhost:1000
 consumers = 137.204.107.188:8000
}
```

Fig. 4. Example of deployment configuration file.

```
1  /* SimpleProdCons organization */
2  org SimpleProdCons implements ProdConsOrg {
3
4    workspace producers {
5      theGui = SimpleGUI ()
6      p1 = SimpleProducer(bufferToUse: theBuffer@consumers)
7             init-task: Producing(numItems: 20)
8      p2 = SimpleProducer(bufferToUse: theBuffer@consumers)
9             init-task: Producing(numItems: 20)
10   }
11
12   workspace consumers {
13     theCounter = Counter(startValue: 0)
14     theBuffer = SimpleBuffer(maxElems: 10)
15     c1 = SimpleConsumer()
16            init-task: Consuming(maxItemsToProcess: 40)
17     c2 = SimpleConsumer()
18            init-task: Consuming(maxItemsToProcess: 40)
19   }
20
21   workspace main {
22     console = Console()
23     monitor = SimpleProductionMonitor
24            init-task: Monitor(buff: theBuffer@consumers)
25   }
26 }
```

Fig. 5. The implementation of a concrete organization.

Finally, the deployment configuration of an organization (details about the deploy process are given in Section V) is managed through a dedicated deployment configuration file. Such file is used in order to specify the binding of the logical workspace-based structure of the organization into the proper set of simpAL nodes targeted for hosting its execution. A simpAL node is a generic machine available on the network, on top of which the simpAL kernel has been properly installed and launched (see Section V for further details). An example configuration file for the `SimpleProdCons` organization is reported in Fig. 4. In this case we distribute the organization on top of three different simpAL nodes (i.e., `137.204.107.188:8000`, `92.10.45.40:8000`, `67.104.7.88:8000`). The model easily allows to swap one configuration file with another giving hence the opportunity to change, even radically, the whole deployment model of a distributed application—e.g., moving, for the same application, from a deployment configuration with all the workspaces hosted on local host to another one in which each workspace is hosted on a different remote simpAL node.

It is up to developers decide how to organize both the logical and physical topology of the application, by taking into account both available hardware resources and application requirements—e.g., an application in which a set of independent tasks are managed by a group of agents and artifacts located in separated and dedicated workspaces can be deployed, to exploits concurrency at its best, either on a set of normal desktops machines or on a server with high computing capabilities.

Applications that do not require to be distributed over the network can simply avoid to specify a deployment configuration file. In this case the simpAL runtime will execute the MAS as a (local) standalone application.

B. Programming the Agents

Analogously to the organization case, the agent programming model separates the definition of the agents' interface from their concrete implementation. For this purpose on the one side is introduced the notion of *roles*, which are used to explicitly define the type of tasks that all the agents that declare to play those roles are capable to do, and on the other side the notion of *agent scripts*, containing the implementation of concrete plans useful to accomplish the tasks related to one or multiple roles.

Fig. 6 shows both an example of definition of role (`Producer`) and an example of a script implementing that role (`SimpleProducer`). A role is identified by a name and includes the definition of a set of task types. Each task type (e.g. `Producing`, lines 3-8) is defined by a name and the declaration of a set of typed parameters (e.g. `maxItems`, line 4), representing information about the task to do. Besides parameters, the definition of a task type can include a set of predefined *attributes* to refine task type specification. Among the others, `understands` makes it possible to specify the beliefs that can be told to agents performing the task (e.g., `Producer` agents can be told about the value of `newItemsToProduce` belief, line 6). So roles define the *type* of agents, used for typing the reference or identifier of an agent. This allows for doing a set of error checking controls at compile time. For instance it is possible verifying, statically, that an agent is assigned only of those tasks which is capable to do or receive only messages that it can understand (because they are both specified in the role the agent is meant to play). It is worth remarking that the notion of role in simpAL is used

to type both local and remote agents references, so this give us the opportunity to perform the above mentioned checks in a transparent manner, without the need to take into the account the actual location of the agents.

A script represents a module of agent behavior, containing both the definition of a set of *plans* useful to accomplish the tasks of the role declared to be implemented by the script, and a set of beliefs that can be accessed by all the plans declared in that script. Beliefs in simpAL have a value and a type—ranging from primitive data types, objects instances of class, or references to specific simpAL abstractions (agents/artifacts, tasks, etc.)[1].

The `SimpleProducer` shown in Fig. 6 has a global belief – `buffer`, used to keep track of the buffer to be used in doing its job – and a couple of plans: one for handling the `Producing` task (lines 23-52) and one for the predefined `Booting` task (lines 19-21), which is executed by default when the script is loaded the first time. Moreover, being the `SimpleProducer` script declared in the context of the `ProdConsOrg` organization model (Fig. 6 line 14), it is possible to directly refer inside the script all the agents and artifacts, both locals and remotes w.r.t the node in which a `SimpleProducer` agent is located, declared in such a model as *literals* (e.g., `console@main` at line 41 and 47, or `theGui@producers` at line 30).

The definition of a plan includes the specification of the type of task for which the plan can be used (e.g. `Producing`, line 23) and a plan body, containing a specification of the procedural knowledge that the agent can use in order to accomplish the task. Such a knowledge can be specified in terms of *action rules*, that are ECA-like rules each specifying an *action* todo along with the event and condition specifying *when* the action must be done. An action rule block – which constitutes the body of a plan, denoted by { ... } – is a set of action rules, possibly including also the definition of local beliefs, i.e. beliefs whose scope is the block. In the most general case, an action rule is of the kind:

`(every-time | when)` *Ev* `:` *Cond* `=>` *Act* `#`*Lbl*

meaning that the specified action `Act` can be executed `every-time` or once that (`when`) the specified event `Ev` occurs and the specified condition `Cond` – which is a boolean expression of the agent beliefs base, including local beliefs – holds. If not specified, the default value of the condition is true. Events, coming transparently from both local and remote workspaces, concern percepts related to either the environment (e.g., the rule at line 40 reacts to the change of `theGUI` artifact `stopPressed` observable property), or messages sent by agents (e.g. line 46), or actions execution, or rather time passing. Actions can be either *internal* – i.e., affecting the agent internal state like a belief assignment or update – or *external*—i.e., or given operations provided by some artifact (e.g. `put`, (line 35) or `println` (line 42 and 48)), or *communicative* actions, to asynchronously send messages to other agents (`tell`, `ask`, `do-task`, `drop-task`, `suspend-task`, etc.). In the former case, such event is immediately enqueued with the execution of the action itself. In the latter case instead, the completion of the actions (with success or failure) may arrive in the future, as an asynchronous event enqueued in the external event queue. The action rule model has been specifically devised to ease the definition of blocks of behavior which may need to integrate and mix the execution of some workflow of actions along with the reactions to some events or condition over the state of the agent (e.g. lines 29-51).

As mentioned before, actions execution can fail, causing

```
1  /* Definition of the Producer role */
2  role Producer {
3    task Producing {
4      maxItems: int;
5      understands {
6        newItemsToProduce: int;
7      }
8    }
9    ...
10 }
11
12 /* Definition of the SimpleProducer script */
13 agent-script SimpleProducer implements Producer
14                                        in ProdConsOrg{
15   /* global beliefs */
16   buffer: Buffer
17
18   /* plans */
19   plan-for Booting (bufferToUse: Buffer) {
20     buffer = bufferToUse
21   }
22
23   plan-for Producing {
24
25     noMoreItemsToProduce: boolean = false;
26     item: int = 0;
27     nItemsTodo: int = maxItems;
28
29     completed-when: noMoreItemsToProduce
30       using:  buffer, theGui@producers {
31
32       /* purely active part */
33       repeat-until items > nItemsTodo {
34         item = item + 1
35         put(item: item) on buffer
36       }
37       noMoreItemsToProduce = true
38
39       /* reactive part */
40       when changed stopPressed in theGui@producers =>
41                                 using: console@main {
42         println(msg: "stopped!")
43         noMoreItemsToProduce = true
44       }
45
46       every-time told newItemsToProduce =>
47                                 using: console@main {
48         println(msg: "new items todo: "+newItemsToProduce)
49         nItemsTodo = nItemsTodo + newItemsToProduce
50       }
51     }
52   }
53 }
```

Fig. 6. Definition of the `Producer` role and of the `SimpleProducer` script.

[1]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 [16]:*"[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"*([16], p. 313).

the generation of proper failures that an agent may perceive as asynchronous events (like in the case of action completion with success), to which it can suitably react. As in previous cases, distribution issues, in this case concerning actions execution and related failures – i.e., external actions involving operation execution over remote artifacts or communications with remote agents – are managed under the hood by the simpAL runtime infrastructure, hence they remain completely transparent from the point of view of the simpAL programming model, which allows, for example, to use the same code to invoke external actions and react to action failures concerning either local or remote components.

Some attributes can be specified for an action rule block to further control its execution and behavior. `using:` specify the list of the identifiers of the artifacts, used inside the block (lines 30, 41, 47). At runtime, when entering a block where an artifact is used, automatically the observable properties of the artifact are continuously perceived and their value is stored in corresponding beliefs in the belief base. Like in the previous cases, also the mapping of observable properties to the agent beliefs is managed transparently by the simpAL runtime infrastructure, shielding the programmer from any possible complexities required for dealing with remote artifacts. The `completed-when:` attribute can be used to specify the condition for which the action rule block execution can be considered completed (line 29).

Agents' behavior is managed by a proper control architecture, that allows for integrating both an active, task-driven and reactive, event-driven behavior and then the *execution cycle* (or *control loop*) that conceptually defines such behavior. Such architecture is inspired to the reasoning cycle of BDI agents, and can be framed here as an extension of the basic event loop found in actors [13]. Conceptually, an agent is a computational entity executing continuously a loop which involves three distinct stages executed in sequence: a *sense* stage, in which the agent fetches percepts (inputs) from the environment – available in an event queue – updating its internal state, a *plan* stage in which, given the current state and the set of current tasks that the agent is actually pursuing, the set of actions to do is selected, and finally an *act* stage in which the selected actions are executed. From a conceptual point of view, an agent is never blocked: it is continuously looping on these stages, possibly without choosing any action to perform if there are no active tasks or there is nothing to do in a specific moment in the tasks it is pursuing.

C. Programming Artifact-Based Environments

The programming model of artifacts is definitely simpler than the agents' one, more similar to the model used for classic passive entities, such as monitors or objects. Artifacts are simple modules 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.

Analogously to the agent case, also for artifact programming we separate the abstract description of the artifact functionalities from their concrete implementation, defining artifact structure and behavior. The former is specified in *artifact models*, defining the usage interface (e.g., Fig. 7 shows the `Buffer` and `Counter` usage interfaces) of all the artifacts implementing that model. Such interface includes *(i)* a set of *operations*, that correspond to the set of actions available to agents for using artifacts (so the repertoire of an agent's actions at runtime depends on the artifacts that the agent knows and can use); and *(ii) observable properties*, as variable-like information items storing those properties of an artifact which may be perceived and exploited by the agents using the artifact. Artifact models are used to define the type of artifacts, used for instance in beliefs on the agent side keeping track of artifacts to be used. Analogously to the agent case, this allows to do a proper set of checks at compile time. For instance, on the agent side, given a typed artifact reference it is possible verifying errors about the actions – i.e., external actions can refer only to artifacts operations specified in the artifact interface (e.g. `put`, Fig. 6 line 35) – and percepts—i.e., belief references related to the observable state of the artifact can refer only to the observable properties defined in the artifact models (e.g., the reference to the `stopPressed` observable property in Fig. 6 at line 40). As in the case of roles, being the notion of artifact model used to type both local and remote artifact references, the static checks just presented apply transparently to any artifact reference, independently from the actual location of the referenced artifact.

```
1  /* Buffer usage interface */
2  interface Buffer {
3    obs-prop nAvailItems: int;
4    operation put (item: int);
5    operation get (?item: int);
6  }
7
8  /* Counter usage interface */
9  interface Counter {
10   obs-prop count: int;
11   operation inc();
12   operation reset();
13 }
```

Fig. 7. Examples of artifact usage interfaces.

The implementation of an artifact is defined in *artifact templates*. The definition of an artifact template includes a name, the declaration of the implemented artifact model, the concrete implementation of operations and the definition of those internal variables that are used in operation implementations. Fig. 8 shows the implementation of a simple buffer and of a counter. Like classes in OOP, artifact templates are a blueprint for creating instances of artifacts. On the agent side, a specific action (`make-artifact`) is available for creating a new artifact, specifying initial parameters and a belief where to store the reference to the artifact created.

Operation behavior is given by a simple sequence of statements, in pure imperative style, using classic control flow constructs, assignment operators, etc. Besides classic statements, specific primitives are introduced to control operation execution. For instance, the `await` statement – used in the `get` and `put` operations – allows for suspending the operation

```
artifact SimpleBuffer implements Buffer {

  /* hidden state variables */
  int maxNumElems;
  java.util.LinkedList<Integer> elems;

  /* constructor */
  init (maxElems: int) {
    count = startValue;
    nAvailItems = 0;
    maxNumElems = maxElems;
    elems = new java.util.LinkedList<Integer>();
  }

  /* operations */
  operation put (item: int) {
    await nAvailItems < maxNumElems;
    elems.add(item);
    nAvailItems = nAvailableItems + 1;
  }

  operation get (?item: int) {
    await nAvailItems > 0;
    nAvailItems = nAvailableItems - 1;
    item = elems.remove();
  }
}
```

```
artifact CounterImpl implements Counter {
  c0: int;

  init (startCount: int) {
    count = startCount;
    c0 = startCount;
  }

  operation inc() {
    count = count + 1
  }

  operation reset(){
    count = c0;
  }
}
```

Fig. 8. Source code of the artifact templates `SimpleBuffer` and `CounterImpl`.

until the specified condition is met (allowing then other operations to be executed). As in the case of monitors, only one operation can be in execution: so if multiple suspended operations can be resumed a certain time, only one is selected. Operation execution in artifacts is *transactional*, in the sense that they are executed in a mutually exclusive way and the changes to the observable state of the artifact (properties) are done atomically. Changes are perceived by agents observing the artifact only when an operation completes (with success).

V. The simpAL Distributed Runtime Infrastructure

The deployment, execution and life-cycle management of programs written in simpAL are in charge of a distributed runtime infrastructure, developed in Java, which has been explicitly devised for managing all this issues transparently with respect to distribution. The background idea that guided the development process of this infrastructure is: the execution and management of distributed MASs should be, from a developer perspective, as simple as the case of centralized, not distributed ones.

To this end the notions of simpAL kernel and simpAL node have been introduced. The former is in charge of the concrete execution of simpAL programs or parts of them (in case of distributed execution). The latter instead is a generic network node in which the simpAL kernel is installed and executed – typically like a demon running in background, launched when the machine boots – used as the basic building block for providing a robust and flexible distributed runtime infrastructure for executing MASs—i.e., a simpAL node can be considered as a generic network node on top of which a user may want to host the execution of simpAL programs, or parts of them.

Once the kernel is up and running in all the interested network nodes, a user can start the execution of a simpAL application by launching it from a generic simpAL node that assumes the role of launch manager. Once the launch starts, the manager, using the information specified into the application deployment configuration file (if any), properly distributes the simpAL program, in its compiled version, among the others target nodes. It is worth remarking that the simpAL runtime infrastructure guarantees the consistency and the alignment of programs' sources either in the case of multiple local launches and in the case of distributed ones—i.e., only the up-to-date version of the compiled sources is loaded or distributed among the interested nodes. Since this phase does not involve any kind of logical dependency, is done in a concurrent manner to speed up the boot process. Then, when the simpAL kernels installed in the interested nodes receive their part of the application, each one of them autonomously starts the booting by properly initializing the workspaces that need to be hosted in that node. During this process, the simpAL kernels communicate with each others using a simple handshake protocol in order to keep track of the current status of the application and to guarantee a proper initialization of the MAS. Finally, only when all the workspaces have been deployed and all the static artifacts contained in them properly created, the agents are spawned and then the MAS can start its execution.

The termination of a running application can be triggered from any of the simpAL nodes in which it is hosted. Once triggered, the termination is managed, exploiting a proper shutdown protocol, in a coordinated manner by all the kernels involved in the shutdown process.

VI. Discussion

Given the presentation of both the simpAL runtime infrastructure and programming model, in this section we provide a critical discussion about some relevant points, in relation to the programming, deployment and management of distributed multi-agent systems in simpAL.

The first one concerns the support given by the simpAL programming model to handle distribution when programming

an application. simpAL has been conceived from the beginning for (also) distributed programming, hence distribution is a feature which is directly part of its agent-oriented programming model. This allows, on the one side to have first class abstractions to conceive and define, in a *explicit manner* during the design phase, the application structure from both a logical and physical point of view. To authors' knowledge, there are not other APLs in the state-of-the-art that give this opportunity. Indeed, usually the application structure is implicit, depicted almost entirely into developers' minds, and it is possible to have a clear picture of it only at runtime when all the MAS's parts have been properly deployed and initialized. On the other side, the programming model makes it possible to get a full *transparency* with respect to distribution when programming agents and artifacts—e.g., the code that one needs to write for implementing an agent communicating with (or spawning) other agents, or an agent using/observing/creating artifacts is the same in spite of the fact that agents/artifacts are in the same workspace (node) or not. A similar transparency level is provided by JADE and by all the APLs in the state-of-the-art that exploit some generic agent platform to enable transparent message-based interactions among local and remote agents (e.g., Jason, AFAPL, etc.). However, usually, transparency related to distribution issues is limited to message exchanges among agents, while instead interactions with other language specific components (e.g. Platform Services in AFAPL) still need to be handled in an ad-hoc manner, taking into account the actual location of such components.

The second point concerns instead the capability to detect programming errors, in particular related to the interactions with distributed components, at compile-time. Indeed, being simpAL a statically typed programming language, it is possible to detect statically, before running the system, a proper set of errors: the referencing of non-existing symbols due to typos, sending to an agent a message that it can not understand, invoking an artifact operation with wrong arguments types, etc. In particular, as described in detail in Section IV, the presence of a proper type system and the transparency provided by the simpAL programming model, give us the opportunity to seamlessly perform a vast set of compile-time error checks concerning the interaction with both local and remote components, without having the burden to deal with distribution issues. A comprehensive description of all the compile-time checks that can be performed in simpAL is outside the scope of the paper. Interested readers can find further details here [18]. Others state-of-the-art APLs provide, in general, a quite weak typing support. Some languages – e.g., Jason, GOAL, AFAPL – do not support at all a notion of type, therefore is not possible perform compile-time errors detection neither for what concerns classical programming errors (e.g., assign to a belief meant to store integer numbers a string value) nor for the ones concerning interactions with distributed components (e.g., requesting the achievement of goals unknown to the agents) [18]. Other ones instead (e.g. JADE), being frameworks realized on top of mainstream OO-based programming languages, exploit the underlying type system for enabling some kind of basic error checks. However, in this APLs part of the abstractions characterizing the agent-oriented paradigm (goals, messages exchanged, etc.) can not be explicitly modeled at the language level (i.e., they are not part of the OO programming model), and therefore this limits the compile-time error checking mechanisms that can be introduced in such languages, in particular the ones related to the interactions with distributed components (e.g., checks for detecting wrong goals assignment to agents).

Finally, the last point we consider, concerns the easy management of the deployment and execution of distributed MASs. We argue that the infrastructural support given by current state-of-the-art APLs makes these tasks quite troublesome, causing developers to manually manage specific deploy/launch procedures that can be instead automatized and taken in charge by APLs' runtime infrastructures. Accordingly, simpAL provides both a programming model and a distributed runtime infrastructure aimed at giving developers the means to manage distributed applications like local ones. However, even if from the one side several efforts have been made in order to engineer a distributed infrastructure that goes beyond the ones provided by state-of-the-art APLs – e.g., no more need to manually deploy / initialize / terminate the different parts of a distributed application, trivial management of different deployment configurations for the same application, etc. – on the other side such infrastructure is still at the prototype level, therefore, it can not be considered as mature and robust as reference agent-based ones, e.g. like the one provided by JADE, used also in industrial contexts.

VII. Ongoing and Future Work

Ongoing and future work concern both topics and arguments which are related to the main contribution of this article, and other ones which are instead outside of this scope and hence, not discussed here—e.g., enriching the task model, definition of social tasks, extending the type system in oder to support sub-typing and inheritance, etc.

First, we want to improve the current implementation of the simpAL runtime platform in order to make it more robust and mature for the execution and management of distributed multi-agent systems. The seamless management of faults that can occur at runtime – e.g., network delays, unexpected shutdown of network nodes, etc. – is a key issue in the general context of distributed systems and related infrastructures that support their execution. So far, in simpAL the only support provided for dealing with faults at runtime is at the programming level, where programmers can deal with the different kind of network problems that can occur by properly reacting to action failures perceived by the agents (see Section IV-B). We aim at improving the current basic support for handling faults at runtime, trying to shield programmers as much as possible from their management. The final objective of this enhancements is the realization of a fault-tolerant runtime infrastructure able to manage in a quite seamless manner: temporary node unreachability, dynamic addition / shutdown of simpAL nodes, workspace migration, etc.

Another main improvement we aim at investigating in the near future concerns the study of a proper extension in order to provide a better support for openness and dynamism in simpAL programs. These are two of the key features of multi-agent systems, giving the opportunity to design applications that can change, adapt and evolve at runtime. Currently simpAL provides a quite weak support for realizing applications able to express this kind of behavior. Indeed, as a design choice, for enabling strict compile-time error checking mechanisms, the set of allowed types in an organization is determined statically (see Section IV). As a consequence, situations involving the interaction with components external to the organization, whose type can be unknown, are not allowed, hence limiting the openness and dynamism of simpAL programs. We want to overcome this weakness by studying a proper extension of the simpAL organizational model, maintaining however our current application model – i.e., an application is seen as an organization structured in term of workspaces – and without renouncing to the possibility of performing compile-time error checks.

Finally, other efforts will be dedicated in the realization and improvement of proper tools for debugging, monitoring, and profiling distributed simpAL programs. Currently simpAL comes along with a simple debugger which allows to inspect agent execution on a step-by-step basis. However this tool is quite simple and, so far, it does not support breakpoint at the artifact/workspace/node level. Therefore, we intend to enrich the current debugger and also develop further tools that will allow to inspect and monitor simpAL programs in a more agile and comprehensive way.

VIII. Conclusion

The availability of proper APLs and related runtime platforms able to support a seamless engineering, deployment and execution of distributed multi-agent systems is a key factor for their successful realization. Accordingly, in this paper we presented simpAL, an agent-oriented programming language providing both a programming model and a distributed runtime platform aiming at making the programming and execution of multi-agent systems quite transparent with respect to their distribution.

A guiding toy example has been used for introducing the support provided by simpAL for dealing with distribution issues at the programming and execution levels, and also for highlighting the main aspects of the simpAL programming model.

Finally, a critical analisys has been provided for pointing out the improvements made in handling distribution issues in simpAL, in relation with the current support given by APLs in the state-of-the-art, and current weaknesses and limitations of both its programming model and distributed runtime platform, along with the plan for their overcome with future work.

References

[1] FIPA Agent Communication Language specification – http://www.fipa.org/repository/aclspecs.html, last retrieved: July 18th 2012.

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

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

[4] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. Special issue: Multi-agent programming. *Autonomous Agents and Multi-Agent Systems*, 23 (2), 2011.

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

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

[7] L. Braubach, A. Pokahr, D. Bade, K. Krempels, and W. Lamersdorf. Deployment of distributed multi-agent systems. *Engineering Societies in the Agents World V*, pages 898–898, 2005.

[8] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[9] T. Finin, R. Fritzson, D. McKay, and R. McEntire. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

[10] K. V. Hindriks. Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications (2nd volume)*, pages 3–37. Springer-Verlag, 2009.

[11] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents™ — summary of an agent infrastructure. In *Proc. of 2nd Int. Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

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

[13] M. Miller, E. Tribble, and J. Shapiro. Concurrency among strangers. In R. De Nicola and D. Sangiorgi, editors, *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer Berlin / Heidelberg, 2005.

[14] 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):432–456, Dec. 2008.

[15] A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, and L. 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.

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

[17] A. Ricci and A. 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.

[18] A. Ricci and A. Santi. Typing multi-agent programs in simpAL. In *Proceedings of the Int. Workshop on Programming Multi-Agent Systems (ProMAS'12)*, Valencia, Spain, 2012.

[19] R. J. Ross, R. W. Collier, and G. M. P. O'Hare. AF-APL - bridging principles and practice in agent oriented languages. In *Programming Multi-Agent Systems*, volume 3346 of *Lecture Notes in Computer Science*, pages 66–88. Springer, 2004.