Externalisation and Internalization: A New Perspective on Agent Modularisation in Multi-Agent System Programming

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

Michele Piunti
DEIS, University of Bologna
Cesena, Italy
Email: michele.piunti@unibo.it

Mirko Viroli
DEIS, University of Bologna
Cesena, Italy
Email: mirko.viroli@unibo.it

Abstract—**Agent modularisation is a main issue in agent and multi-agent system programming. Existing solutions typically propose some kinds of constructs – such as *capabilities* – to group and encapsulate in well-defined modules *inside* the agent different kinds of agent features, that depend on the architecture or model adopted—examples are goals, beliefs, intentions, skills. In this paper we introduce a further perspective, which can be considered complimentary to existing approaches, which accounts for *externalizing* some of such functionalities into the computational environment where agents are (logically) situated. In this perspective, agent modules are realised as suitably designed *artifacts* that agents can dynamically exploit as *external* tools to enhance their action repertoire and – more generally – their capability to execute tasks. Then, to let agent (and agent programmers) exploit such capabilities abstracting from the low-level mechanics of artifact management and use, we exploit the dual notion of *internalization*, which consists in dynamically consulting and automatically embedding high-level usage protocols described in artifact *manuals* as agent plans. The idea is discussed providing some practical examples of use, based on CArtAgO as technology for programming artifacts and Jason agent platform to program the agents.**

I. Introduction

Agent modularisation is a main issue in agent-oriented software engineering and multi-agent system (MAS) programming, accounting for devising proper structures and mechanisms to modularise agent behaviour, enhancing maintainability, extensibility and reuse of agent-based software. Existing solutions – which are briefly surveyed in Section IV – typically propose constructs that make it possible to group, encapsulate and reuse in well-defined modules agent features, that can vary according to the architecture or model adopted: for instance, modularisation in BDI agents have been proposed in terms of capabilities [3], [2], goals [17], intentions [8], to mention some.

In all existing approaches modules are components *inside* agents. In this paper we introduce a further complimentary perspective, which accounts for improving modularity by *externalizing* some functionalities into the *computational environment* where agents are (logically) situated, as external facilities that agents exploit as tools extending their capabilities.

The background of this idea is given by the research work on *environment design and programming* in MAS [19], [15], in which the computational environment where agents are situated is considered a *first-class abstraction* that can be suitably designed and programmed so as to improve MAS engineering, encapsulating functionalities that concern, for instance, agent interactions, coordination and organisation.

In this context, CArtAgO [15], [16] – which will be exploited in this paper – has been proposed as a general-purpose framework and infrastructure for building shared computational worlds that agents, possibly belonging to heterogeneous agent platforms and written using different agent programming languages [14], can exploit to work together. Being based on the A&A (Agents and Artifacts) meta-model [15], [12], [16], CArtAgO's computational environments are modelled as set of distributed workspaces, containing dynamic sets of *artifacts*.

The artifact abstraction is a key concept on which is the contribution of this paper is based. From the agent viewpoint, artifacts are first-class entities of agents' world, representing resources and tools that agents can dynamically instantiate, share and use to support individual and collective activities. From the MAS designer viewpoint, artifacts are useful to uniformly design and program those abstractions inside a MAS that are not suitably modelled as agents, and that encapsulate functions to be exploited by individual agents or the overall MAS—for instance mediating and empowering agent interaction and coordination, or wrapping external resources. CArtAgO provides a concrete computational and programming model for artifacts [15], composed by a set of Java-based API to program artifacts on the one side, and agent API to work inside artifact-based environment on the other side.

The availability of artifact-based computational environments in multi-agent system programming makes it possible to enrich the strategies for modularising agents by exploiting artifacts as modules that can be dynamically instantiated/used/composed, extending the basic repertoire of agent actions and capabilities. So, instead of being wrapped into modules inside agents – either structuring the agent program or extending the agent architecture – such capabilities are *externalised* into artifacts that agents can use and exploit as personal – and in some cases shared – tools.

In this paper we develop this idea, providing some practical examples using CArtAgO and Jason agent programming language. It is important to remark that this approach is not meant to replace existing proposals, but to be integrated with them. On the one side, some agent features are clearly not externalisable, or – at least – it is not useful to externalise them. For instance, for cognitive agents, capabilities concerning deliberation or the manipulation of the internal mental state. On the other side, the approach allows for properties which are not typically provided by existing proposals. For instance, the reuse of the same kind of module (artifacts) across different agent programming languages and platforms.

The remainder of the paper is organised as follows: in Section II we describe in detail the idea, providing some examples to clarify the approach. Then, in Section III we introduce *internalization* as a key mechanism layered on top of externalisation that allows agents and agent programmers to exploit functionalities externalized in artifacts abstracting as much as possible from the low-level mechanics of artifact management and use. In Section IV we provide an overview of existing works on agent modularisation and how the contribution of this paper is related to them. Finally, in Section V we provide concluding remarks, sketching current limitations and the next steps planned to further develop of the idea.

II. Externalisation: Agent Modules Implemented as Artifacts

A. The Basic Idea

The basic idea is to exploit artifacts as modules to encapsulate new capabilities for agents, in particular extending the repertoire of agent actions with the set of operations provided by artifacts[1]. We call this *externalisation* since the capabilities of an agent are not extended by acting agent internal architecture or program, but by extending the set of external resources and tools (artifacts) that the agent can use to do its work.

By applying externalisation, a module is conceived as a tool that the agent may eventually create and use by need. In particular: artifact operations encapsulate the functionalities that would be provided by module actions; artifact usage interface and observable properties (and events) represent the module interface; the non-observable state of the artifact is used to implement the hidden inner state of the module; and finally the *manual* of an artifact can be used to store the description of high-level usage protocols accompanying the module—this point will be discussed in detail in Section III.

[1]the main features of the artifact abstraction are extensively described in [12], [16], [15]. Briefly, each artifact has a usage interface listing a set of usage interface controls that can be used to trigger and control the execution of operations inside the artifact. By executing operations, an artifact can generate observable events (signals) that can be perceived both by the agent using the artifact and by all those that are focussing (observing) it. Besides observable events, an artifact can have observable properties, whose value (and changes) are automatically perceived by all the observing agents

At runtime (execution time) the burden of the execution of modules is no more on the agent side, like in the case of modules implemented as components inside agents, but on the artifact side: artifact operations are executed asynchronously by independent control flows, managed by the CArtAgO machinery. The agent can control operations execution by means of the usage interface of the artifact, perceiving its state and results in terms of observable properties and events. This has a strong impact on efficiency at runtime: (a) agents do not waste time and computational resources for the execution of the processes related to the module functionalities; (b) the approach transparently exploits the concurrency and parallelism support provided by the underlying execution environment.

Then, module management is mapped on to artifact creation/disposal/discovery, in particular module activation is given by instantiating the artifact or by locating an existing one; module composition is realised by using multiple artifacts. Actually, the approach supports also a kind of module *sharing* by exploiting artifacts shared and co-used simultaneously by multiple agents: this can be very useful for supporting effective and efficient forms of agent coordination (Subsection II-E).

In the following, we clarify and discuss the idea by describing some concrete examples of increasing complexity, exploiting CArtAgO to implement artifacts and Jason [1] to program agents exploiting artifacts as modules. It's worth noting that here we use Jason, but an analogous discussion would be for other agent programming languages, such as 2APL, or platforms like Jadex. The examples are classified along two basic dimensions: the state dimension – differentiating between state-less and state-full modules (tools) – and the sharing dimension – differentiating between personal and shared modules (tools).

B. *Modules as* State-less *Tools*

The simplest kind of module is given by a library of *internal actions* which are meant to be exploited by agents as functions, extending the basic computing capabilities provided by default by the agent language.

As a concrete example, suppose to extend agents with some math capabilities not directly supported by the language—let's take the *sine* function as a simple example. Languages like Jason allow for solving the problem by extending the agent architecture, with the implementation of new internal actions exploiting the Java-based API provided by the Jason platform. Externalisation makes it possible to solve the problem without the need of extending directly agents, by programming a new kind of artifact – functioning as a *calculator* tool in this case – that the agent can instantiate and (re-)use by need. Fig. 1 shows a sketch of its implementation in CArtAgO API and of a Jason agent exploiting the functionality[2]. The action module `sin(+Value,?Result)` is implemented by the `computeSin(+Value)` operation of the artifact, and

[2]Details about the artifact abstraction and CArtAgO API, as well as Jason and their integration, are outside the scope of this paper: the interested reader can found them in literature [15], [12], [16], [14], [1]

```
package tools;

public class Calculator extends Artifact {

  @OPERATION void computeSin(double x){
    signal("sin",x,Math.sin(x));
  }
  @OPERATION void computeCos(double x){
    signal("cos",x,Math.cos(x));
  }
  @OPERATION void computeSqrt(double x){
    if (x >= 0){
      signal("sqrt",x,Math.sqrt(x));
    } else {
      signal("math_error");
    }
  }
  ...
}
```

```
// Jason agent using its calculator

!doComputations.

+!doComputations
  <- ?mytool("tools.Calculator",Id);
     cartago.use(Id,computeSin(1.57),s0);
     cartago.sense(s0,sin(1.57,Y));
     cartago.use(console,
       println("The sin value of 1.57 is ",Y)).

+?mytool(ToolType,Id)
  <- .my_name(AgName);
     .concat(AgName,"-",ToolType,ToolName);
     cartago.makeArtifact(ToolName,ToolType,Id);
     +mytool(ToolType,Id).
```

Fig. 1. *(Left)* A `Calculator` artifact encapsulating math functionalities. `computeSin` operation, once triggered, generates an observable event of the type `sin(X,Y)` which is then perceived by the agent using the calculator. *(Right)* A Jason agent exploiting the calculator: the first time the calculator is used it is created, using a conventional name given by concatenation of the agent name and the artifact type.

action execution is realised in terms of a sequence of basic CArtAgO actions to interact with it. In particular, the agent first retrieves the tool identifier – eventually creating the artifact if it is the first time it is used and keeping track of such identifier by a `mytool` belief; then, it triggers the execution of the operation on the tool (by means of the `use` CArtAgO primitive) and then exploits a sensor to perceive the result (by means of the `sense` CArtAgO primitive). The result is represented by an observable event `sin(X,Y)` generated by the `signal` primitive executed in the `computeSin` operation. It's worth remarking that the computation of the sine value is done asynchronously w.r.t. the agent activities: synchronisation occurs when the agent inspects the sensor to perceive the result.

This first example – in spite of its simplicity – is useful to give a taste of the approach: instead of extending the agent architecture by means of new (internal) `sin` action, in this case the extension is achieved by means of en external calculator tool that an agent can instantiate and use. Being externalised into an artifact, the functionalities can be exploited by any kind of agent whose platform has been integrated with CArtAgO—besides Jason, other examples include Jadex, 2APL, AgentFactory: so the approach promotes extensibility and reusability across heterogeneous agent languages and platforms.

C. *Modules as* State-ful *Tools*

Then, besides state-less modules, artifacts can be useful to encapsulate functionalities involving a state and providing actions working with such state, functioning as personal stateful tools. The state can be either an observable part, i.e that can be considered part of the agent knowledge, or a hidden part of the module. The observable part is mapped onto artifact observable properties, which are then perceived by the agent observing the artifact as percepts (mapped into beliefs in cognitive architectures).

As an example, consider the `Calculator2` artifact depicted in Fig. 2, providing a support for executing a sequence of operations, keeping track and making it observable the updated result by means of the `result` observable property and providing functionalities to undo the operations. Fig. 2 shows an example of an agent using the calculator, adding repeatedly a value (3.0) by "pressing the button" `add` until it perceives that the result is greater than 10. After that, it restores the previous result (which is 9, in this case) and prints it on the console.

This second example shows how the approach supports the management of observable information of the module on the one side and *information hiding* on the one side: inner structures needed to realise the module functionalities – such as the list of the partial results, to enable undo and redo in the example – are implemented by artifact inner data structures, which are accessed and changed by artifact operations.

D. *Modules Wrapping External Actions*

In previous examples we considered modules encapsulating sets of *internal* actions: besides these ones, modules can be also devised so as to extend agents with capabilities to access/interact with external resources (such as data-base), including resources to communicate with external systems (such as network channels, GUIs). In that case, the externalisation perspective accounts for implementing such modules as tools wrapping the access and interaction with those external resources, hiding as much as possible the low-level details related to the use of the resources and providing the agent a high-level interface for exploiting the functionalities. Actually CArtAgO provides – as an auxiliary library – a basic set of artifact types that can be exploited to this end, including tools for working with ODBC data-bases, for using socket-based network channels, and for creating and managing graphical user interfaces. Examples of this kind of tools can be found in CArtAgO distribution—not reported here for lack of space.

E. *Modules Wrapping Mechanisms for Interaction, Coordination, Organisation*

Agent coordination is a main issue in multi-agent system programming; direct communication models – including approaches based on speech-act based conversations – are not always the most effective solution to achieve agent coordination

```
public class Calculator2 extends Artifact {
  Stack<Double> results;

  @OPERATION void init(){
    defineObsProperty("result",0);
    results = new Stack<Double>();
  }
  @OPERATION void add(double x){
    double res = getObsProperty("result").doubleValue();
    results.push(res);
    updateObsProperty("result",res + x);
  }
  @OPERATION void sub(double x){
    double res = getObsProperty("result").doubleValue();
    results.push(res);
    updateObsProperty("result",res - x);
  }
  @OPERATION void undo(){
    if (!results.isEmpty()){
      updateObsProperty("result",results.pop());
    } else {
      signal("result_stack_empty");
    }
  }
}
```

```
!doComputations.

+!doComputations \
  <- ?mytool("tools.Calculator2",Calc);
     cartago.focus(Calc);
     !doSums(Calc).

+!doSums(Calc): result(X) & X<=10
  <- cartago.use(Calc,add(3.0),s0);
     cartago.sense(s0,op_exec_completed("add"));
     !doSums(Calc).

+!doSums(Calc): result(X) & X>10
  <- cartago.use(Calc,undo,s0);
     cartago.sense(s0,op_exec_completed("undo"));
     cartago.observeProperty(Calc,result(Y));
     cartago.use(console,print("Final value: ",Y)).
```

Fig. 2. *(Left)* State-full extension of the Calculator, exploiting observable properties. *(Right)* Jason agent exploiting the calculator.

```
public class Semaphore extends Artifact {
  int count;

  @OPERATION void init(int startCount){
    count = startCount;
  }

  @OPERATION(guard="isFree") void acquire(){
    count--;
  }

  @GUARD boolean isFree(){
    return count > 0;
  }

  @OPERATION void release(){
    count++;
  }
}
```

```
!doJob.

+!doJob
  <- !locateTool("tools.Semaphore","cs",[1],Tool);
     !work(Tool).

+!work(Tool)
  <- cartago.use(Tool,acquire);
     !doMyCriticalTask(0);
     cartago.use(Tool,release);
     !work(Tool).

+!doMyCriticalTask(C) : C < 10
  <- .println(C); .wait(10); !doMyCriticalTask(C+1).
+!doMyCriticalTask(10).

+!locateTool(Type,Name,Args,Id) : not tool_avail(Name)
  <- cartago.lookupArtifact(Name,Id).
-!locateTool(Type,Name,Args,Id) : not tool_avail(Name)
  <- +~tool_avail(ToolName); !locateTool(Type,Name,Args,Id).
+!locateTool(Type,Name,Args,Id) :  ~tool_avail(Name)
  <- cartago.makeArtifact(Name,Type,Args,Id).
-!locateTool(Type,Name,Args,Id) :  ~tool_avail(Name)
  <- -~tool_avail(Name); !locateTool(Type,Name,Args,Id).
```

Fig. 3. (Left) A Semaphore artifact, that can be exploited as a tool for extending agents with basic synchronization capabilities. (Right) Jason agent exploiting the Semaphore for realising critical sections

and various kinds of interaction-oriented and coordination-oriented *mechanisms* can be devised to this end. From an agent programming language perspective, the implementation of these mechanisms typically accounts for extending the basic agent language with a specific set of primitives tailored to provide some kind of coordination/organisation functionalities. This strongly reminds *coordination languages* [7], which are orthogonal to classical computational languages (such as C, Java, Prolog) and extend them with basic coordination primitives to enable communication and synchronisation. Linda is a well-known example of coordination language [6]. Actually, in the context of multi-agent systems this occurs also for organisation: languages such as J-MOISE [9], for instance, extends the basic Jason language with MOISE organisational primitives.

This case is similar to the previous one, since such primitives can be considered external actions involving some kind of inter-actions with other agents. By adopting externalisation, such capabilities can be encapsulated in proper artifacts, extending agents with coordination capabilities without the need of extending the agent language. Differently from the previous cases, these artifacts are meant to be *shared* by the agents – as a kind of shared modules – providing operations enabling and ruling the interaction among the agents exploiting them.

As a simple example, consider here the problem of extending an agent with the capability of executing *critical sections*, which require the coordination of all the agents running in the same environment. To this end, we can simply use a semaphore artifact functioning as a shared lock by the agents, providing two basic operations: to acquire it – to be used in the prologue of the critical section – and to release it – to be used in the epilogue of the section. Fig. 3 shows on the left the semaphore artifact and on the right an agent using it to realise a critical section. In this case all agents interact with the same artifact—called `cs` in the example. From an agent (and

```
usageprot compute_sin {
  :function sin(X,Y)
  :body {
     locateMyTool(ToolId);
     freshSensor(S);
     use(ToolId,computeSin(X),S);
     sense(S,sin(X,Y)).
  }
}
```

```
!doComputations
  <- !setup;
     !doTheJob.

+!doTheJob
  <- cartago.consultManual("tools.Calculator");
     cartago.consultManual("tools.Console").

+!doTheJob
  <- !sin(1.57,Y);
     !print("The sin value of 1.57 is ",Y).
```

Fig. 4. (Left) A usage protocol defined in the `Calculator` manual (Right) Jason agent exploiting the manual to use the `Calculator`

agent programmer) perspective, this can be seen as a facility extending the basic agent coordination capabilities, alternative to the use of communication protocols. The interested reader can find more complex examples of coordination tools in CArtAgO distribution: among the other the `TupleSpace` artifact, which in the perspective of this paper can be framed as a module extending agents with the Linda coordination language.

III. Internalization: Using Artifacts as Agent Modules

Actually, a main problem of externalisation is the level of abstraction adopted for allowing an agent to access and exploit the new capabilities provided by the modules: when programming agents exploiting modules externalised into artifacts, the programmer must specify the details related to artifacts use and management. We tackle this problem by means of *internalization*.

Internalization accounts for introducing a proper abstraction layer which makes it possible to exploit artifacts functionalities in terms of agent actions, abstracting as far as possible – from an agent programmer point of view – from the low-level mechanics of artifact management and use. This can be achieved by exploiting artifacts *manual*. Being a feature of the basic artifact abstraction, the manual is that document providing a machine-readable description – written by the artifact developer – of artifact functionalities and operating instructions [12], [18]. Such information are meant to be dynamically read, interpreted and *internalized* by the agent, embedding such a knowledge in terms of proper *plans* about how to use the artifacts of that type and when.

Here we focus on the operating instructions, as that part of the manual describing *usage protocols*, i.e. high-level plans encapsulating sequences of low-level operations to be executed in order to exploit artifact functionalities. In current model, a usage protocol is characterised by a *function*[3], which defines the functionality to be exploited, a *precondition*, defining the condition under which the functionality can be exploited, and a *body*, as a sequence of actions. Fig. 4 shows an example of usage protocol defined in the manual for the calculator, to exploit the sine function. A simple first-order logic-based language is used to define the protocols: the complete syntax and semantics of the language is not reported here for lack of space, we describe the language informally by means of concrete examples.

The function is specified by means of `:function` tag and is represented by a logic term, possibly containing parameters detailing input and output (in terms of unbounded variables) information characterising the function. In the calculator example, `sin(X,Y)` is the function of the usage protocol to compute the sine function. The function of a usage protocol is directly linked to agent goals: a usage protocol with a function $func$ is mapped into agent plan(s) that are triggered to achieve goals matching $func$, according to some kind of matching function that depends on the agent architecture adopted. In the case of Jason agents, for instance, the usage protocol is triggered to achieve goals of the type `sin(X,Y)`: in the example (Fig. 4, on the right) this happens by means of the `!sin(1.57,Y)` action.

The precondition can be specified by the `:precond` tag and is represented by a logic expression specifying the conditions that must hold concerning either the function parameters or agent beliefs[4] (which typically can include the state of the observable properties of the artifact). If missing, the default value of the expression is `true`. Preconditions are used in the second example (Fig. 5), showing the manual of the semaphore artifact described in Subsection II-E, providing high-level usage protocols to execute critical sections. In particular, two alternative protocols are specified for entering a critical section, one to be used when the agent is not already inside the critical section and the other one in the opposite case. The belief `inside_cs(`*`ToolId`*`)` – added by one protocol when the entering succeeds – is used to distinguish this case.

The body – specified by means of the `:body` tag – contains a sequence of actions, including basic CArtAgO actions (`use`, `sense`, `focus`, etc.), auxiliary actions to locate artifacts and internal actions for inspecting and updating the belief and goal base of the agent. From a syntactical point of view, `;` is used as sequence operator, `+`*`Bel`* and `-`*`Bel`* is used to add and remove beliefs and `.` to indicate the end of the plan.

On the agent side, two further actions are provided respectively for consulting and forgetting the content of a manual, `consultManual(ArtifactTypeName)` and `forgetManual(ArtifactTypeName)`[5]. By consulting

[3]The term “function” here must be interpreted as “functionality”, so not related to functional programming languages or mathematical functions

[4]The notion of “belief” can be replaced here with “knowledge” for agent programming languages not having that concept

[5]the parameter does not refer to the name of a specific existing artifact, but to the name of an artifact type, which must be available by current workspace

```
usageprot enter_critical_section1 {
  :function enterCS
  :precond not inside_cs(_)
  :body {
    locateTool("Semaphore","cs",ToolId);
    use(ToolId,acquire); +inside_cs(ToolId).
  }}

usageprot enter_critical_section2 {
  :function enterCS
  :precond inside_cs(_)
  :body {}}

usageprot exit_critical_section {
  :function exitCS
  :precond inside_cs(ToolId)
  :body {
    use(ToolId,release); -inside_cs(ToolId).
  }}
```

```
!doJob
  <- !setup;
     !work.

+!setup
  <- cartago.consultManual("tools.Semaphore");
     cartago.consultManual("tools.Console").

+!work
  <- !enterCS;
     !doMyCriticalTask(0);
     !exitCS;
     !work(Tool).

+!doMyCriticalTask(C) : C < 10
  <- !println(C);
      .wait(10);
      !doMyCriticalTask(C+1).
+!doMyCriticalTask(10).
```

Fig. 5. (Left) Usage protocols defined in the `Semaphore` manual for doing critical sections (Right) Jason agent executing critical sections exploiting the usage protocols

the manual, the practical knowledge contained inside is fetched and translated into agent local plans, which are triggered by achievement goals which have the same signature of the function.

The key point here is that the agent programmer has not to be aware and explicitly code the usage protocol, which is specified – instead – by artifact developers: s/he must simply know the interface of the usage protocol, in terms of the function and beliefs involved. So the approach promotes a strong separation of concerns and finally more compact agent programs. This is exemplified by the source code of the Jason agent in Fig. 5, whose behaviour is analogous to the one in Subsection II-E but where `!enterCS` and `!exitCS` are the only lines of code that the agent programmer has to write to let the agent enter and exit a critical section.

IV. Related Works

Agent modularisation is a main open issue in agent programming languages and various solutions have been proposed in literature.

In [3], the notion of capability has been introduced and implemented in the JACK commercial Java-based multi-agent framework. Capabilities represent a cluster of components of a BDI agent, both encapsulating beliefs, events and plans and promoting global meta-level reasoning over them. From a software engineering perspective – which is the main perspective of this paper – capabilities enable software reuse, being building blocks that can be reused in different agents. This notion of capability is further refined and improved in Jadex, a Java and XML based BDI agent platform [2]. Capabilities are here generalised and extended so as to support an higher degree of reusability, devising a mechanism that allows for designing and implementing BDI agents as a composition of configurable agent modules (capabilities) which are treated as black-boxes exporting interfaces in line with object-oriented engineering principles.

A somewhat different but related idea of modularisation is discussed in [11], in which a modular BDI agent programming architecture is proposed, mainly targeted at supporting the design of specialised programming languages for single agent development, and at providing transparent interfaces to existing mainstream programming languages for easy integration with external code and legacy software. The proposed architecture is independent to the internal structure of its components and agent reasoning model, and uses interaction rules to define the connections between the design components. This draws a clear distinction between knowledge representation issues and their dynamics, and promotes the design and development of specialized programming languages.

A goal-oriented approach to modularisation for cognitive agent programming languages is proposed in [17], suggesting agent goals as the basis of modularisation. The approach is then discussed providing a formal semantics in the context of the 3APL agent programming language. A similar notion has been proposed in the agent language GOAL [8] where a module is a component within an agent encapsulating policy-based intentions to be triggered in a particular situation. This approach combines the knowledge and skills to adequately pursue the goals of the agent in that situation and is used to realize a mechanism to control nondeterminism in agent execution.

A role-based approach to modularisation and reuse has been proposed in the context of AgentFactory agent platform and ALPHA programming language. To engender code reuse the framework makes use of the notion of commitments and role template [4].

Finally, to authors' knowledge the most recent approaches to modularity have been introduced in the 2APL and Jason agent platforms. In the former, similarly to the other related works, a module is considered as an encapsulation of cognitive components. The added value of authors' approach is the introduction of set of generic programming constructs that can be used by an agent programmer to perform a variety of operations on modules, giving agent programmers full control in determining how and when modules are used. In that way modules can be used to implement a variety of agent concepts such as agent role and agent profile [5]. The latter proposes a mechanism for modular construction of Jason agents from functionally encapsulated components – containing beliefs, goals and plans – so as to improve the support of the language for the development of complex multi-agent systems, in an agent-oriented software engineering perspective [10].

In all these approaches modules are components *inside* agents. In this paper we explored a dual perspective, which allows for implementing modules as components *outside* the agents, externalised in proper tools and artifacts that agents can exploit (and possibly share) for their tasks. This allows for fruitfully integrated the approach described in this paper with existing ones, promoting a strong separation of concerns in programming agents, using – on the one side – agent language/architecture and related module mechanisms to define and modularise only those aspects that strictly concern agent internal aspects (state update and action selection in general, deliberation and means/ends reasoning in cognitive architectures); on the other side, artifact-based computational environments to engineer and modularise all those resources and tools that agents may exploit to achieve their tasks.

V. Conclusion and Future Works

In this paper we discussed a novel perspective to deal with agent modularisation in multi-agent system programming, based on the availability of artifact-based computational environments. The approach is not meant to be alternative to existing approaches, but rather a complimentary strategy which aims at improving the level of reusability, maintainability, extensibility – including dynamic extensibility – of multi-agent-based software systems.

Starting from this basic idea, now several points need to be further developed. The basic externalisation model must be improved so as to manage aspects related to protection: for instance, devising a strategy to prevent agents to access personal tools (modules) of other agents. Then, the language adopted to define the usage protocols, described in Section III, currently does not tackle some main problems that are important in the practice: two main ones are the management of name clashes (between the function and beliefs defined by the protocol and existing plans/goals/beliefs of the agents or other usage protocols) and the management of failures (currently mapped tout-court onto agent plan failure). Also no formal semantics has been devised yet. These points are part of future works. Also, the model currently adopted to describe usage protocols – in terms of function, preconditions, and a body – can be considered just a first step: some other further features will be explored, such as the possibility to define – besides preconditions – also *invariant* conditions, stating the conditions that must hold for all the duration of the usage protocol, and post-conditions, i.e. conditions that must hold when the protocol has completed. Besides conditions expressing the correctness of the protocol, tags could be used to support the reasoning about the tools (modules), such as an `effect` tag to specify the expected state of the artifact(s) and of agent beliefs by successfully executing the protocol, towards a truly cognitive use of artifacts/modules [13], [18].

Finally, in order to validate the approach, we plan to identify specific domains/applications to make it clear the advantages of externalisation/internalization, eventually integrating different cognitive agent programming languages/platforms besides Jason, such as 2APL and Jadex.

References

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

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

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

[4] R. Collier, R. R. Ross, and G. M. O'Hare. Realising reusable agent behaviours with ALPHA. In *Multiagent System Technologies*, volume 3550 of *LNCS*, pages 210–215. Springer, 2005.

[5] M. Dastani, C. Mol, and B. Steunebrink. Modularity in agent programming languages: An illustration in extended 2APL. In *Proceedings of the 11th Pacific Rim International Conference on Multi-Agent Systems (PRIMA 2008)*, volume 5357 of *LNCS*, pages 139–152. Springer, 2008.

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

[7] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96, 1992.

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

[9] R. H ubner, JomiFred Bordini and G. Picard. Jason and MOISE+: Organisational programming in the agent contest 2008. In *Dagstuhl Seminar on Programming Multi-Agent Systems*, volume 08361, 2008.

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

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

[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] M. Piunti, A. Ricci, L. Braubach, and A. Pokahr. Goal-directed interactions in artifact-based MAS: Jadex Agents playing in CArtAgO environments. In *Proceedings of Intelligent Agent Technology 2008 (IAT '08)*. IEEE/ACM, 2008.

[14] A. Ricci, M. Piunti, L. D. Acay, R. Bordini, J. Hubner, and M. Dastani. Integrating artifact-based environments with heterogeneous agent-programming platforms. In *Proceedings of 7th International Conference on Agents and Multi Agents Systems (AAMAS08)*, 2008.

[15] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in CArtAgO. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications, Vol. 2*, pages 259–288. Springer, 2009.

[16] A. Ricci, M. Viroli, and A. Omicini. The A&A programming model & technology for developing agent environments in MAS. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Programming Multi-Agent Systems*, volume 4908 of *LNAI*, pages 91–109. Springer, 2007.

[17] M. B. van Riemsdijk, M. Dastani, J.-J. C. Meyer, and F. S. de Boer. Goal-oriented modularity in agent programming. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1271–1278, New York, NY, USA, 2006. ACM.

[18] M. Viroli, A. Ricci, and A. Omicini. Operating instructions for intelligent agent coordination. *The Knowledge Engineering Review*, 21(1):49–69, Mar. 2006.

[19] D. Weyns, A. Omicini, and J. J. Odell. Environment as a first-class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, Feb. 2007. Special Issue on Environments for Multi-agent Systems.