Multi-Agent Programming

– Integrating Intelligent Systems with MAS –

O. Boissier

Univ. Lyon, IMT Mines Saint-Etienne, LaHC UMR CNRS 5516, France

CPS2 M1 – Fall 2020

Integrating Intelligent Systems

Multi-Agent Oriented Programming & JaCaMo

Outline

Integrating MAOP and Existing Technologies

Introduction

How to integrate MAOP and the supporting technologies – JaCaMo in this case – with existing libraries and technologies, based on other paradigms?
Integration can be in two directions:

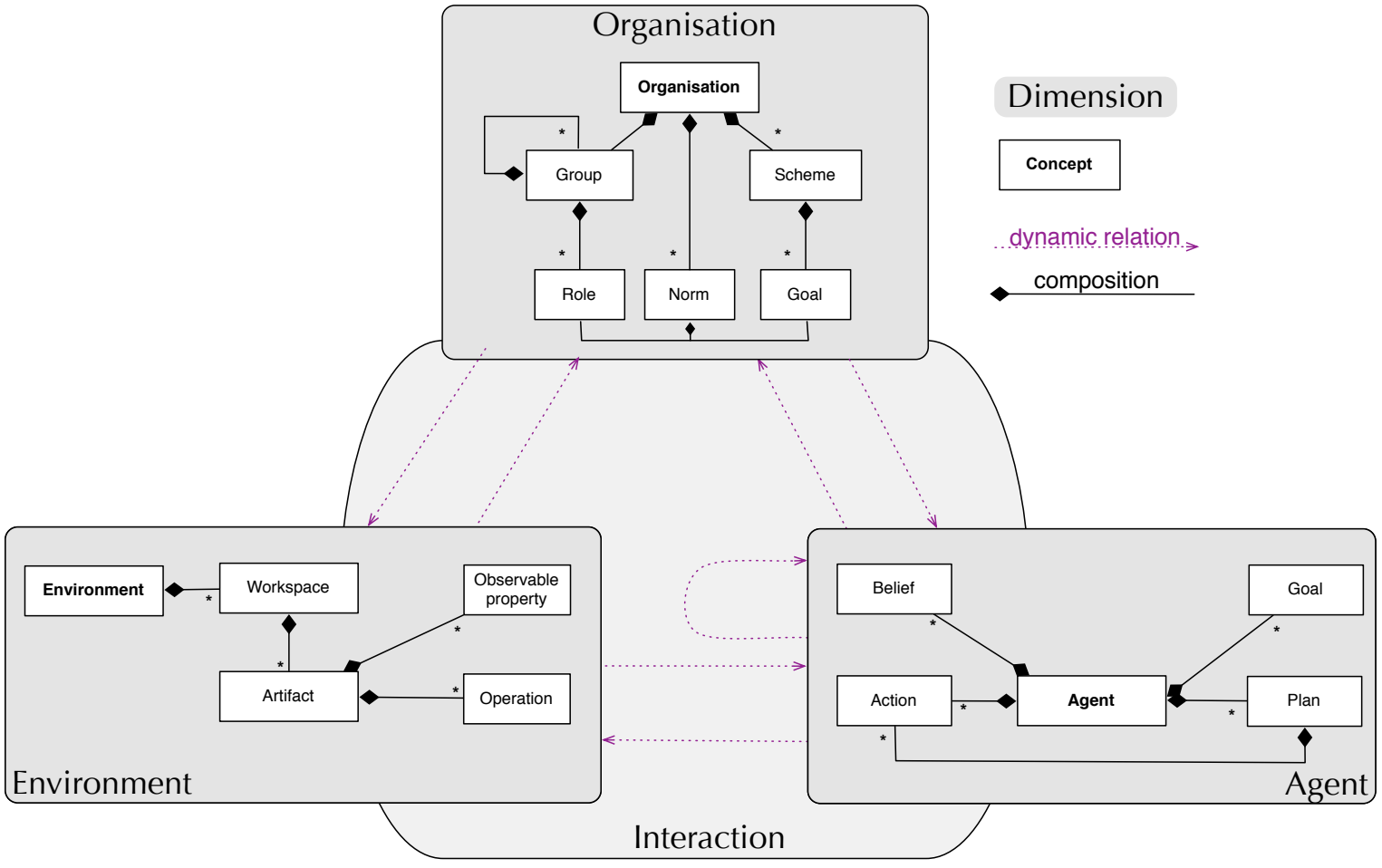
- either embedding inside the MAS program some existing technology such as a library or a framework,
- or integrating the MAS inside some existing platform.

In what follows, we will:

- Discover guidelines about integration depending on the specific technology to integrate
- Discuss some cases in specific application domains

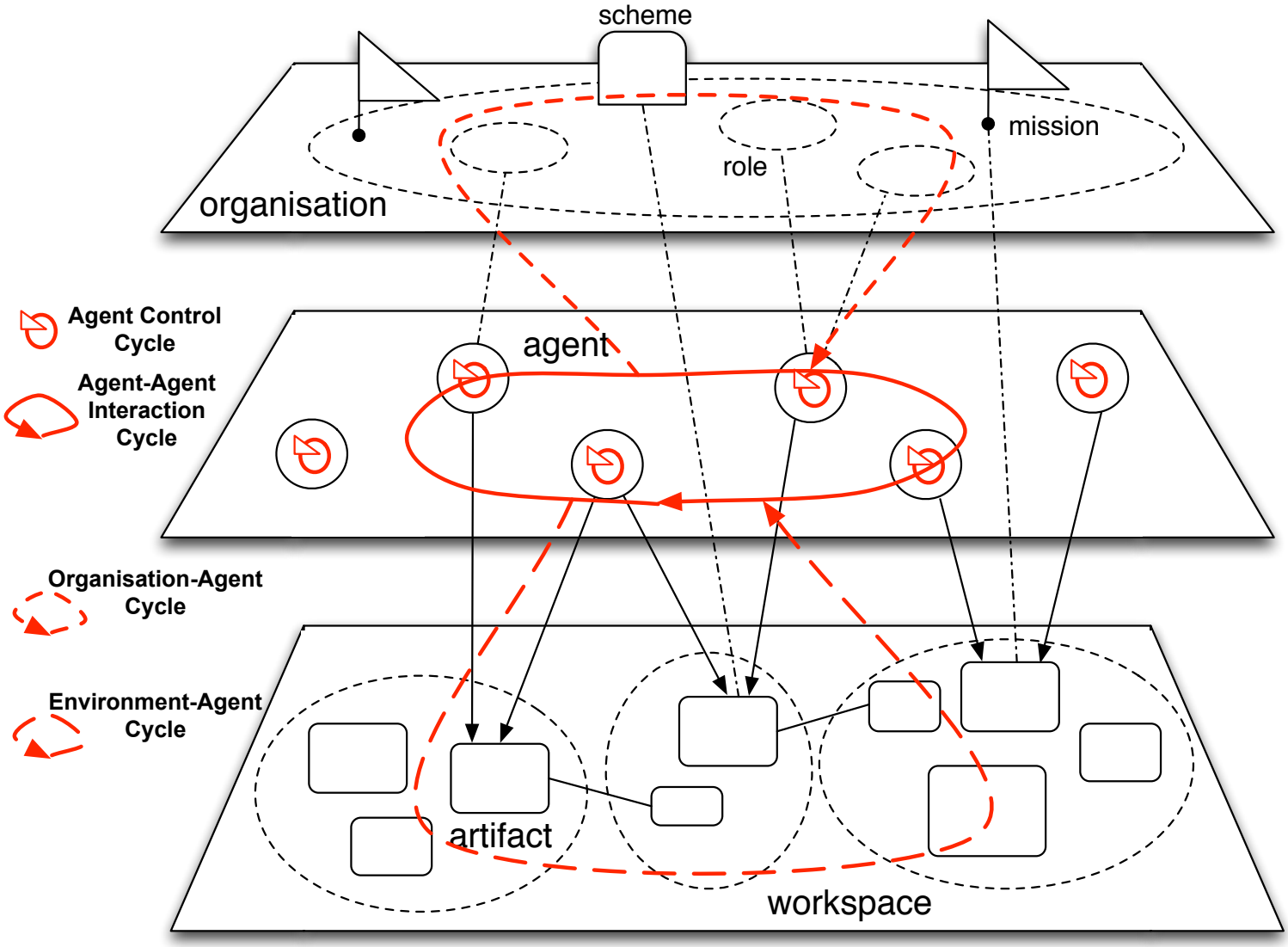
MAOP: Concepts and Relations

Introduction



MAOP: Dynamics

Introduction



What to integrate?

Introduction

- **Library**: functionality provided by some module exposing some kind of interface/API, without introducing any **control** issue.
  - Example: library for effectively managing JSON data objects.
- **Software framework**: reusable software environment meant to facilitate the development of software applications, by means of generic functionality that can be changed by additional user-written code. Differently from simple libraries, software frameworks typically introduce also some kind of **control** architecture, defining how the application is executed.
  - Example: frameworks for developing GUI-based applications (JavaFX), mobile apps (Android), event-driven asynchronous applications vert.x.
- **Application Platform**: software framework that provide a core technology on which software developers can build programs for some specific platform.

Two Ways for Integration using MAOP

Introduction

Two ways to integrate existing libraries and frameworks - eventually written in mainstream programming languages:

- **Agent extension**: the integration is realised by extending the agents' capabilities customising either their architecture or their set of internal actions to wrap the technology to be integrated.
  $\rightsquigarrow$ an agent can exploit the technology as its new personal capability.

- **Artifact embedding**: the integration is realised by designing and implementing new artifacts wrapping the technology to be integrated.
  $\rightsquigarrow$ an agent can exploit and interact with the integrated technologies as resources/tools part of their environment, possibly to be shared with other agents.

Agent extension vs. Artifact embedding

Introduction

From a technical point of view, there are important differences about how the code is executed when using either internal actions or artifacts.

- With artifact embedding, the code, wrapped into operations, is executed by a thread of control of the environment runtime, asynchronously.

- With agent extension, the code is executed directly by the thread of control of the reasoning cycle, in a synchronous way.

Agent extension vs. Artifact embedding (contd)

- Agent extension has better performance: no overhead caused by context switches and the computational cost is equivalent to a simple (method) call.
- Agent extension of functionalities that could be heavy/long term from a computational point of view may have impact on the reasoning cycle execution and then on the reactivity of the agent to perceive events
  $\rightsquigarrow$ use artifact embedding, offloading on the artifacts of computational load.
- Artifact embedding is more convenient when the library to be integrated can be more useful as a tool which could be shared and concurrently used by multiple agents or (possibly stateful) tools embedding long-term heavy computations.

Outline

Integrating using Agent Extension

Integrating Libraries

Agent extension is straightforward when the functionality is about some state-less function that extends the agent capabilities.

- Ex: internal action json_to_list to parse JSON data into a list:

```
package json_tools;
public class json_to_list extends DefaultInternalAction {
    public Object execute(TransitionSystem ts,
                Unifier un, Term[] args) throws Exception {
        ... // the code that implements the internal action
    }}
```

- On the agent side:

```
test_json <- json_tools.json_to_list(
      "{ \"name\": \"Sofia\", \"age\": 11 }", L);
      .println(L). // the list L is [ name("Sofia"), age(11)
```

Integrating using Artefact Embedding

Integrating Libraries

Enrich the agent capabilities with the ability to compute the Fast-Fourier-Transform, to convert and analyse a signal from its original domain (time or space) to a representation in the frequency domain.

- Library effectively implementing the algorithm (e.g. open-source Apache Commons Mathematics Library can be quite computational expensive
- Libraries can be either based on Java or JVM-based languages or even other languages, exploiting the Java Foreign-Function-Interface (FFI) API to integrate them.
- The library can be packaged as a FFTCalculator artifact, providing operations to make the transformation, making the result available as action feedback of a transform operation and/or as observable property of the artifact.

Integrating using Artefact Embedding (contd)

Integrating Libraries

```
public class FFTCalculator extends Artifact {
  private FastFourierTransformer trf;
  void init(){
    trf = new FastFourierTransformer(DftNormalization.STANDARD);
  }
  @OPERATION void fftTransform(double[] f,
                               OpFeedbackParam<Complex[]> r) {
    Complex[] res = trf.transform(f,TransformType.FORWARD);
    r.set(res);
  }
  // aux function to manage data structures
  @OPERATION void getReal(Complex[] data,
                          OpFeedbackParam<Double[]> r) { ... }
  @OPERATION void getComplex(Complex[] data,
                             OpFeedbackParam<Double[]> r){ ... }
}
```

Integrating using Artefact Embedding (contd)

Integrating Libraries

An agent creates a FFTCalculator and uses it to compute the Fast Fourier Transform given an array of double and prints on the console the result, as a list of complex numbers (represented by the Complex class).

```
+!test_fft
  <- makeArtifact("calc","FFTCalculator",[]);
     cartago.new_array("double[]",[5.0, -2.0, 1.0, 2.0], Data);
     fftTransform(Data, Res);
     !print_result(Res). // printing: ( 6 )( 4 + 4i )( 6 )( 4-4i )

+!print_result([]).
+!print_result([V|T]) <-
     cartago.invoke_obj(V,getReal,Re);
     cartago.invoke_obj(V,getImaginary,Im);
     !print_complex(Re,Im);
     !print_result(T).

+!print_complex(Re,0) <- print("( ", Re, " )").
+!print_complex(0,Im) <- print("( ", Im, "j )").
+!print_complex(Re,Im) : Im < 0 <- print("( ",Re," - ",-Im,"i )").
+!print_complex(Re,Im) <- print("( ", Re, " + ",Im, "i )").
```

Java Objects on the Agent Side

Integrating Libraries

- JaCaMo provides a set of internal actions packaged inside the cartago library
- The internal actions to create and manipulate Java objects are part of the CArtAgO framework, referred as JavaLibrary that makes it possible to
  - instantiate Java objects (cartago.new_obj);
  - instantiate arrays (cartago.new_array);
  - invoke methods on objects (cartago.invoke_obj).
- Detailed information about the JavaLibrary can be found in the JaCaMo documentation.

Integrating Libraries with their own Threads

More complex libraries could exploit asynchronous programming to provide some of their functionalities, using under-the-hood some threads of control to that purpose.

- The artifact API beginExtSession/endExtSession makes it possible for such external threads to safely:
  - change the state of an artifact,
  - change the state of observable properties,
  - generate signals to model asynchronous events.

Outline

Integrating Frameworks

- The management of callbacks and interaction with external threads is frequent when integrating frameworks/platforms, which may enforce the full control architecture of the application.
- Artifact embedding makes it possible to have a clear separation between the specific execution logic provided by a framework and MAS execution.
- The artifact provides a high-level interface for agents to work with the framework (e.g. main window in case of a GUI-based application).
- The use of an observable property enables to keep track and make observable to agents the state of the application (e.g. "pressed" or "not_pressed" for a button)
- The artifact functions as a bridge to the framework, implementing (and hiding with respect to the MAS) the machinery to make the framework working.
- The external session is used to update the observable state of the artifact.

Outline

Integrating within Platforms

Integration of the MAS:

- either as a component of the platform,
- or by executing the MAS on a different process and implementing components inside the platform that function as bridges exploiting some Inter-Process Communication (IPC) mechanism (such as sockets).

MAS as a Component of the platform:

- Use of the JaCaMo API jacamo.infra.JaCaMoLauncher for spawning programmatically a MAS

```
public class LaunchMAS {
    public static void main (String args[]) throws Exception {
       jacamo.infra.JaCaMoLauncher.main(
                        new String[]{ "test-mas.jcm" });
    }}
```

MAS as a different process:

- Use of boundary artifacts to implement the bridge between the systems, embedding/hiding the use of Inter-Process-Communication mechanisms and protocols to interact with the external platform.

Bibliography I