Programming Adaptive Microservice Applications: an AIOCJ Tutorial

Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, Maurizio Gabbrielli

7

Programming Adaptive Microservice Applications: an AIOCJ Tutorial*

Saverio Giallorenzo[1], Ivan Lanese[1],
Jacopo Mauro[2], and Maurizio Gabbrielli[1]

[1] *Focus Team, University of Bologna/INRIA, Italy*
[2] *Department of Informatics, University of Oslo, Norway*

*Supported by the COST Action IC1201 BETTY, by the EU project FP7-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems*, by the GNCS group of INdAM via project *Logica, Automi e Giochi per Sistemi Auto-adattivi*, and by the EU EIT Digital project *SMAll*.

Programming Adaptive Microservice Applications: an AIOCJ Tutorial

Abstract

This tutorial describes AIOCJ, which stands for *Adaptive Interaction Oriented Choreographies in Jolie*, a choreographic language for programming microservice-based applications which can be updated at runtime. The compilation of a single AIOCJ program generates the whole set of distributed microservices that compose the application. Adaptation is performed using adaptation rules. Abstractly, each rule replaces a pre-delimited part of the program with the new code contained in the rule itself. Concretely, at runtime, the application of a rule updates part of the microservices that compose the application so to match the behavior specified by the updated program. Thanks to the properties of choreographies, the adaptive application is free from communication deadlocks and message races even after adaptation.

Keywords: Adaptation, Distributed Applications, Microservices, Correctness-by-Design.

1 Introduction

Today, most applications are distributed, involving multiple participants scattered on the network and interacting by exchanging messages. While still widely used, the standard client-server topology has shown some of its limitations and peer-to-peer and other interaction patterns are raising in popularity in many contexts, from social networks to business-to-business, from gaming to public services. Programming the intended behavior of such applications requires to understand how the behavior of the single program of one of their nodes combines with the others, to produce the global behavior of the application. In brief, it requires to master the intricacies of concurrency and distribution. There is clearly a tension between the *global* desired behavior of a distributed application and the fact that it is programmed by developing *local*

programs. Choreographies [1, 2, 3, 4, 5], and more specifically choreographic programming [6], aim at solving this tension by providing to developers a programming language where they directly specify the global behavior. A sample choreography that describes the behavior of an application composed of one `client` and one `seller` is:

```
1   product_name@client = getInput( "Insert product name" );
2   quote: client( product_name ) -> seller( sel_product )
```

The execution starts with an action performed by the `client`: an input request to the local user (line 1). The semicolon at the end of the line is a sequential composition operator, hence the user input should complete before execution proceeds to line 2. Then, a communication between the `client` and the `seller` takes place: the `client` sends a message and the `seller` receives it. A more detailed description of the choreographic language used in the example above is presented in Section 3.

Following the choreographic programming approach, given a choreography, the local programs that implement the global specification are automatically generated by the language compiler, ready for the deployment in the intended locations and machines. For instance, the compilation of the choreography in the example produces the local codes of both the `client` and the `seller`. The local code of the `client` starts with a user interaction, followed by the sending of a message to the `seller`. The local code of the `seller` has just one action: the reception of a message from the `client`.

The choice of a choreographic language has also the advantage of avoiding by construction common errors performed when developing concurrent and distributed applications [7]. Notably, these include communication deadlocks, which may cause the application to block, and message races, which may lead to unexpected behaviors in some executions.

Another advantage of the choreographic approach is that it eases the task of adapting a running distributed application. We recall that nowadays applications are often meant to run for a long time and should adapt to changes of the environment, to updates of requirements, and to the variability of business rules. Adapting distributed applications at runtime, that is without stopping and restarting them, and with limited degradation of the quality of service, is a relevant yet difficult to reach goal. In a choreographic setting, one can simply specify how the global behavior is expected to change. This approach leaves to the compiler and the runtime support the burden of concretely updating the code of each local program. This update should be done avoiding misbehaviors

while the adaptation is carried out and ensuring a correct post-adaptation behavior.

This tutorial presents AIOCJ[1], which stands for *Adaptive Interaction Oriented Choreographies in Jolie*, a framework including *i*) a choreographic language, AIOC, for programming microservice-based applications which can be dynamically updated at runtime and *ii*) its runtime environment. The main features of the AIOCJ framework are:

Choreographic approach: the AIOC language allows the programmer to write the behavior of a whole distributed application as a single program;

Runtime adaptability: AIOCJ applications can be updated by writing new pieces of code embodied into AIOC *adaptation rules*. Adaptation rules are dynamically and automatically applied to running AIOCJ applications, providing new features, allowing for new behaviors, and updating underlying business rules.

Microservice architecture: AIOCJ applications are implemented as systems of microservices [8]. Indeed, we found that the microservice architectural style supports the fine-grained distribution and flexibility required by our case. As a consequence, AIOCJ applications can interact using standard application-layer protocols (e.g., SOAP and HTTP) with existing (legacy) software thus also facilitating and supporting the integration of existing systems.

A more technical account of the AIOCJ framework can be found in the literature, describing both the underlying theory [9] and the tool itself [10]. AIOCJ can be downloaded from its website [11], where additional documentation and examples are available.

2 AIOCJ Outline

As described in the Introduction, AIOC is a choreographic language for programming microservice-based applications which can be dynamically updated at runtime. The AIOCJ framework is composed of two parts:

- the AIOCJ Integrated Development Environment (IDE), provided as an Eclipse plugin, that lets developers write both AIOC programs and

[1] The tutorial refers to version 1.3 of AIOCJ.

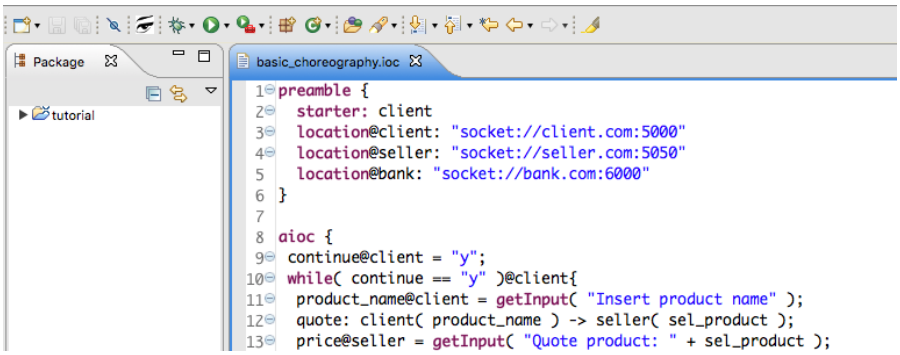

Figure 1 The AIOCJ IDE

the adaptation rules that change the behavior of AIOCJ applications at runtime;
- the AIOCJ Runtime Environment (RE), which is used to support the execution and the adaptation of AIOCJ applications.

The AIOCJ IDE (see the screenshot in Figure 1) offers standard functionalities such as syntax highlighting and syntax checking. However, the most important functionality of the IDE is the support for code compilation. The target language of the AIOCJ compiler is Jolie [12, 13], the first language natively supporting microservice architectures. A key feature of the Jolie language is its support for a wide range of communication technologies (TCP/IP sockets, Unix domain sockets, Bluetooth) and of protocols (e.g., HTTP, SOAP, JSON-RPC) that makes it extremely useful for system integration. AIOC inherits this ability since it makes the communication capabilities of Jolie available to the AIOC programmer.

Since AIOC is a choreographic language, each AIOC program defines a distributed application. The application is composed of different nodes, each taking a specific *role* in the choreography. Each role has its own local state, and the roles communicate by exchanging messages. The structure of AIOCJ applications makes the compilation process of AIOCJ peculiar for two main reasons:

- the compilation of a single AIOC program generates one Jolie microservice for each role involved in the choreography, instead of a unique executable for the whole application;
- the compilation may involve either an AIOC program, or a set of AIOC adaptation rules. In particular, the latter may be compiled even after the

compilation, deployment, and launch of the AIOC program. Thus AIOC adaptation rules can be devised and programmed while the application is running, and therefore applied to it at runtime.

Adaptation rules target well-identified parts of AIOC programs. Indeed, an AIOC program may declare some part of its code as adaptable by enclosing it in a `scope` block. Abstractly, the effect of the application of an AIOC adaptation rule to a given `scope` is to replace the `scope` block with new code, contained in the adaptation rule itself. Concretely, when the distributed execution of an AIOC program reaches a `scope`, the AIOCJ RE checks whether there is any adaptation rule applicable to it. If this is the case, then the running system of microservices adapts so to match the behavior specified by the updated choreography. This adaptation involves coordinating the distribution and execution of the local codes corresponding to the global code in the adaptation rule. If instead no rule applies, the execution proceeds as specified by the code within the `scope`.

In the rest of this section we describe the architecture of AIOCJ and the workflow that developers, or better DevOps[2], have to follow in order to compile, deploy, and adapt at runtime an AIOCJ application (a more detailed step-by-step description is in Section 6). We instead dedicate Sections 3 to 5 to the description of the AIOC language.

2.1 AIOCJ Architecture and Workflow

The AIOCJ runtime environment comprises a few Jolie microservices that support the execution and adaptation of compiled programs. The main microservices of the AIOCJ runtime environment are:

- Adaptation Manager, a microservice in charge of managing the adaptation protocol;
- Adaptation Server, a microservice that contains a set of adaptation rules;
- Environment, a microservice used to store values of global properties related to the execution environment. These properties may be used to check whether adaptation rules are applicable or not.

More precisely, a runtime environment includes one Adaptation Manager, zero or more Adaptation Servers, each of them enabling a set of adaptation rules, and, if needed to evaluate the applicability conditions of the rules,

[2] DevOps is a portmanteau of "development" and "operations" used to indicate the professional figure involved in the development, deployment, and operation of the application.

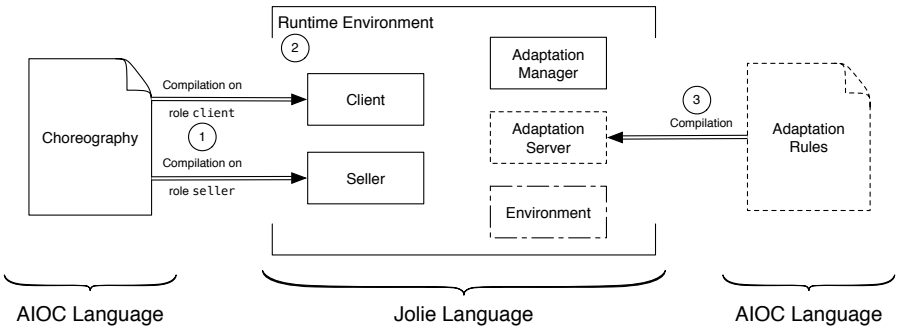

Figure 2 The AIOCJ framework — deployment and execution of a choreography.

one Environment microservice. Adaptation Servers can be added or removed dynamically, thus enabling dynamic changes in the set of rules.

Microservices compiled from AIOC code interact both among themselves, as specified by the choreography, and with the Adaptation Manager, to carry out adaptation. Indeed, when a `scope` is about to be executed, the Adaptation Manager is invoked to check whether the `scope` can be executed as it is, or if it must be replaced by the code provided by some adaptation rule, made available by an active Adaptation Server. In fact, when started, an Adaptation Server registers itself at the Adaptation Manager. The Adaptation Manager invokes the registered Adaptation Servers to check whether their adaptation rules are applicable. In order to check applicability, the corresponding Adaptation Server evaluates the *applicability condition* of the rule, possibly interacting with the Environment microservice. The first applicable adaptation rule, if any, is used to replace the code of the `scope`.

Let us consider an example. Take a simple choreography in AIOC involving two roles, `client` and `seller`. Figure 2 depicts the process of compilation ① and execution ② of the AIOC. From left to right, we use the IDE to write the AIOC and to compile it into a set of executable Jolie microservices (Client and Seller). To execute the generated application, we first launch the Adaptation Manager and then the two compiled microservices.

Now, let us suppose that we want to adapt our application. Assuming that the choreography has at least one `scope`, we only need to write and introduce into the system a new set of adaptation rules. Figure 2 depicts the needed steps. From right to left, we write the rules (outlined with dashes) and we compile them using the IDE ③. The compilation of a set of adaptation rules in AIOCJ produces a single Adaptation Server (also outlined with dashes). After the

compilation, the generated Adaptation Server is deployed and started, and it registers itself at the Adaptation Manager. If environmental information is needed to evaluate the applicability condition of the rule, then the DevOps has also to deploy the Environment microservice. From now on, until the Adaptation Server is shut down, the rules it contains are active and can be applied to the application. Actual adaptation happens when a `scope` is about to execute, and the applicability condition of the rule for the current `scope` is satisfied. This adaptation is performed automatically and it is completely transparent to the user, except for possible differences in the visible behavior of the new code w.r.t. the original one.

3 Choreographic Programming

The main idea of choreographic programming is that a unique program describes the behavior of a whole distributed application. The main construct of such a program are *interactions*, such as:

```
quote: client( product_name ) -> seller ( sel_product )
```

This interaction specifies that role `client` sends a message to role `seller` on operation `quote`. The value of the message is given by the evaluation of expression `product_name` (here just a variable name) in the local state of role `client`. The message will be stored by the `seller` in its local variable `sel_product`. An interaction involves two roles of the choreography, but other choreography constructs involve just one role. For instance, an assignment like `continue@client = "y"`, means that the string `"y"` is assigned to the variable `continue` of role `client`, as specified by the `@` operator.

Let us now detail a simple AIOC program implementing a `client`/`seller` interaction featuring a payment via a `bank` (see Listing 7.1). We will use this program as running example throughout the tutorial. Lines 1–6 form the `preamble`, which specifies some deployment information:

- line 2 declares the `starter` of the choreography, i.e., the first role that needs to be started and the one that coordinates the start of the application by waiting for the other roles to join;
- lines 3–5 specify how the roles participating to the choreography can be reached. In this case, all the three roles communicate using TCP/IP sockets, as specified by the `"socket://"` prefix of the URI.

The actual code is introduced by the keyword `aioc`. After the local assignment at line 9, line 10 introduces a while loop. The `@client` suffix specifies

```
1  preamble {
2   starter: client
3   location@client: "socket://client.com:5000"
4   location@seller: "socket://seller.com:5050"
5   location@bank: "socket://bank.com:6000"
6  }
7
8  aioc {
9   continue@client = "y";
10  while( continue == "y" )@client{
11   product_name@client = getInput( "Insert product name" );
12   quote: client( product_name ) -> seller( sel_product );
13   price@seller = getInput( "Quote product: " + sel_product );
14   if ( price > 0 )@seller{
15    quoteResponse: seller( price ) -> client( product_price );
16    accept@client = getInput(
17    "Do you accept to buy the product: " + product_name +
18     " at price: " + product_price + "? [y/n]" );
19    if ( accept == "y" )@client{
20     orderPayment: client( product_price ) -> bank( amount );
21     authorisePayment@bank = getInput(
22     "Do you authorise the payment: " + amount + " [y/n]?" );
23     if ( authorisePayment == "y" )@bank{
24      issuePayment: bank( amount ) -> seller( payment );
25      productDelivery: seller() -> client();
26      r@client = show( "Object delivered" )
27     } else {
28      r@client = show( "Payment refused" )
29     }
30    }
31   } else {
32    _r@client = show( "Product " + product_name + " unavailable." )
33   };
34  continue@client = getInput( "Continue shopping? [y/n]" )
35  }
36 }
```

Listing 7.1 Running example: basic choreography.

that the guard is evaluated by the `client` in its local state. Notice that the decision about whether to enter the loop or not is taken by the `client` but it impacts also other roles. These roles are notified of the choice by auxiliary communications which are automatically generated. The assignment at line 9 and the while loop starting at line 10 are composed using a semicolon,

which represents sequential composition. Line 11 is again an assignment, where built-in function `getInput` is used to interact with the local user. The function creates a window showing the string in parameter and returns the input of the user. Line 12 is an interaction between the client and the seller. The next interesting construct is at line 14, featuring a conditional. As for while loops, the conditional specifies which role is in charge of evaluating the guard, and other roles are automatically notified of the outcome of the evaluation. Function `show` (line 26) is a built-in function like `getInput`, simply showing a message.

Abstracting from the technical details, the choreography specifies that the `client` asks the quote for a product (line 12), and then decides whether to buy it or not (line 19). In the first case, the `client` asks the `bank` to perform the payment (line 20). If the payment is authorized (line 23), then the money is sent to the `seller` (line 24), which delivers the product to the `client` (line 25). At the end of the interaction, the `client` may decide to buy a new product or to stop (line 34).

When writing AIOC programs, beyond the usual syntactic errors, one should pay attention to a semantic error peculiar of choreographic programming. Indeed, a semicolon specifies that the code before the semicolon should be executed before the code after the semicolon. However, since there is no central control, such a constraint can only be enforced if for each pair of statements `S` and `T` such that `S` is just before the semicolon and `T` is just after the semicolon, there is a role occurring in both `S` and `T`. This property is called connectedness [9] and it is needed to enforce the sequentiality of the actions. When connectedness does not hold, AIOCJ IDE alerts the user by showing the error "The sequence is not connected". Instead of asking the programmer to satisfy connectedness, one could extend AIOCJ to automatically insert auxiliary communications to ensure connectedness, similarly to what is done for while loops and conditionals. Such an extension is left as future work.

4 Integration with Legacy Software

The example in the previous section shows how one can program a distributed application in AIOCJ. However, such a distributed application is closed: there is no interaction between the application and the outside world, except for basic visual interactions with the users of the application. As we will see below, AIOCJ applications are not necessarily closed systems. Indeed, AIOCJ provides a strong support to integration with legacy software. We

```
include quoteProduct from "socket://localhost:8000" with SOAP
include makePayment from "socket://localhost:8001/IBAN" with HTTP

preamble {
 starter: client,
 location@client: "socket://client.com:5000"
 location@seller: "socket://seller.com:5050"
 location@bank: "socket://bank.com:6000"
}

aioc {
 continue@client = "y";
 while( continue == "y" )@client{
  product_name@client = getInput( "Insert product name" );
  quote: client( product_name ) -> seller( sel_product );
  price@seller = quoteProduct( sel_product );
  if ( price > 0 )@seller{
   quoteResponse: seller( price ) -> client( product_price );
   accept@client = getInput(
   "Do you accept to buy the product: " + product_name +
    " at price: " + product_price + "? [y/n]" );
   if ( accept == "y" )@client{
    orderPayment: client( product_price ) -> bank( amount );
    authorisePayment@bank = makePayment( amount );
    if ( authorisePayment == "y" )@bank{
     issuePayment: bank( amount ) -> seller( payment );
     productDelivery: seller() -> client();
     r@client = show( "Object delivered" )
    } else {
     r@client = show( "Payment refused" )
    }
   }
  } else {
   _r@client = show( "Product " + product_name + " unavailable." )
  };
 continue@client = getInput( "Continue shopping? [y/n]" )
 }
}
```

Listing 7.2 Running example: integration with external services.

already cited that AIOCJ is based on the microservice technology. As such, it supports interaction with external services via standard application-layer protocols, such as SOAP and HTTP. Such services are seen as functions inside AIOC programs, and can be invoked and used inside expressions.

Let us see how this can be done by refining our running example from Listing 7.1 into the one in Listing 7.2.

In Listing 7.2, lines 1 and 2 declare two external services, `quoteProduct` invoked using SOAP and `makePayment` invoked using HTTP (more precisely, a POST request carrying XML data). Both external services communicate with AIOCJ using TCP/IP sockets. The first service is invoked at line 16 by the `seller` and it is used to check the price of a given product. In principle, such a service can be either publicly available or a private service of the `seller`. Here, we assume that this service gives access to the `seller` IT system, e.g., to the database storing prices of the available products. The second service is invoked at line 24 by the `bank`, and gives access to the `bank` IT system. One can easily imagine to make the example more realistic by adding other external services taking care, e.g., of shipping the product.

We now discuss in more detail how function arguments are encoded for service invocation and how the result is sent back to the caller. In general AIOCJ functions can have an arbitrary number of parameters, separated by commas. The parameters are embedded in a tree structure which is then encoded according to the chosen application-layer data protocol. The tree structure has an empty root with a number of children all named `p` (for parameter) carrying the parameters of the invocation, in the order in which they are specified. The return value instead has basic type (such as string, integer, double) and it is contained in the root of the response message.

For instance, consider a sample function `myFunction`, with three parameters, a string, an integer, and a double. If the data protocol for `myFunction` is SOAP, then the AIOCJ application would send a SOAP message as reported in Listing 7.3. A possible reply to the message above is a SOAP message of the form reported in Listing 7.4.

Other application-layer data protocols would produce similar structures. Currently, AIOCJ supports SOAP, HTTP, SODEP (i.e., Jolie's binary data protocol), JSON/RPC, and XML/RPC. As far as the communication medium is concerned, AIOCJ supports other options beyond TCP/IP sockets, namely Bluetooth with URIs of the form `"btl2cap://0050CD00321B:101"` and Unix domain sockets with URIs of the form `"localsocket://var/comm/socket"`. The choice of the communication medium and the choice of the application-layer data protocols are orthogonal.

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <myFunctionRequest>
      <p xsi:type="xsd:string">parameter1</p>
      <p xsi:type="xsd:int">2</p>
      <p xsi:type="xsd:double">3.14</p>
      </myFunctionRequest>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 7.3 Function invocation: SOAP message request.

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <myFunctionResponse xsi:type="xsd:string">
      responseValue
    </myFunctionResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 7.4 Function invocation: SOAP message response.

5 Adaptation

We now come to the main feature of AIOCJ, namely the support for adaptation. Adaptation is performed in two stages:

1. when writing the original AIOC program, one should foresee which parts of the code could be adapted in the future, and enclose them into `scope`s;
2. while the AIOC program is running, one should write adaptation rules to introduce the desired new behavior.

We introduce in Listing 7.5 three `scope`s to show how adaptation can be enabled in the running example in Listing 7.2.

```
include quoteProduct from "socket://localhost:8000" with SOAP
include makePayment from "socket://localhost:8001/IBAN" with HTTP

preamble {
 starter: client
 location@client: "socket://client.com:5000"
 location@seller: "socket://seller.com:5050"
 location@bank: "socket://bank.com:6000"
}

aioc {
 continue@client = "y";
 while( continue == "y" )@client{
  product_name@client = getInput( "Insert product name" );
  quote: client( product_name ) -> seller( sel_product );
  price@seller = quoteProduct( sel_product );
  if ( price > 0 )@seller{
   quoteResponse: seller( price ) -> client( product_price );
   accept@client = getInput(
   "Do you accept to buy the product: " + product_name +
    " at price: " + product_price + "? [y/n]" );
   if ( accept == "y" )@client{
    orderPayment: client( product_price ) -> bank( amount );
    authorisePayment@bank = makePayment( amount );
    if ( authorisePayment == "y" )@bank{
     scope @seller {
      issuePayment: bank( amount ) -> seller( payment );
      productDelivery: seller() -> client()
     } prop { N.scopename = "transaction-execution" };
     scope @seller {
      r@client = show( "Object delivered" )
     } prop { N.scopename = "success-notification" }
    } else {
     scope @seller {
      r@client = show( "Payment refused" )
     } prop { N.scopename = "failure-notification" }
    }
   }
  } else {
   _r@client = show( product_name + " is unavailable." )
  };
 continue@client = getInput( "Continue shopping? [y/n]" )
 }
}
```

Listing 7.5 Running example: enabling adaptation.

Scope `transaction-execution` at lines 26–29 encloses the body of the business transaction, with the idea that this can be changed to support integration with a shipper service, or more refined payment protocols. Then, we have two `scope`s, `success-notification` (lines 30–32) and `failure-notification` (lines 34–36), which are in charge of notifying the `client` of the outcome of the transaction, with the idea that different forms of notification, e.g., via e-mail or SMS, could be implemented in the future. Developers can equip `scope`s with properties describing their nature and characteristics. These properties can be used to decide whether a given rule should apply to a given `scope` or not. In the example, we just use a property `scopename` to describe each `scope`. In general, however, many properties can be used. For example, if some `scope` encloses a part of the code which is critical for security reasons, one of its properties could declare the security level of the current code. Such a declaration is under the responsibility of the programmer and it is in no way checked or enforced by the AIOCJ framework.

Note that each `scope` is followed by an annotation `@`*role* that declares the coordinator of the adaptation procedure of the `scope`. The coordinator is in charge of invoking the Adaptation Manager, which handles the selection of an applicable adaptation rule. The Adaptation Manager can access the internal state of the coordinator to check whether an adaptation rule is applicable or not. The coordinator is also in charge of fetching the local codes compiled from the selected adaptation rule and of distributing them to the other roles.

Remark 1 *We highlight that there is no precise convention on how to place* `scope`*s: one should try to foresee which parts of the AIOC program are likely to change. As a rule of thumb, parts which are critical for security or performance reasons may be updated to improve the security level or the performance of the application. Parts which instead implement business rules may need to be updated to change the business rules. Finally, parts that manage interactions with external services may need to be updated to match changes in the external services. There is also a trade-off involved in the definition of* `scope`*s. On the one hand, large* `scope`*s are rarely useful, since they could be updated only before the beginning of their execution, which can be quite early in the life of the application. On the other hand, small* `scope`*s may be problematic, since a meaningful update may involve many of them and currently AIOCJ does not provide a way to synchronize when and how* `scope`*s are updated.*

Now that the application in Listing 7.5 is equipped with scopes, it is ready to be deployed, and offers built-in support for adaptation. While the

```
rule {
 on { N.scopename == "transaction-execution" and
      E.split_payment_threshold < price }
 do {
  issuePayment: bank( amount / 2 ) -> seller( first_payment );
  productDelivery: seller() -> client();
  issuePayment: bank( amount / 2 ) -> seller( second_payment );
  payment@seller = first_payment + second_payment
 }
}
```

Listing 7.6 Adaptation rule: split payment.

application is running, a new need may emerge. Assume for instance that the application, meant for trading cheap products, needs to be used also for more expensive ones. In this previously unforeseen setting, the fact that the payment is performed in a single installment and before the shipping of the product may be undesirable for the `Client`. One can meet this new need by providing an adaptation rule (see Listing 7.6) where the payment is performed in two installments, each consisting in half of the original `amount`: one sent before and the other after the delivery of the product. This rule targets `scope`s with property `scopename` equal to `transaction-execution` and it applies only if the `price` of the product is above a `split_payment_threshold` available in the Environment microservice. The idea is that such a threshold may be agreed upon by the `client` and the `seller` or established by some business regulation. We remark that properties of the `scope`, like `N.scopename`, are prefixed by `N` while values provided by the Environment microservice, like `E.split_payment_threshold`, are prefixed by `E`. Names with no prefix refer to variables of the role that coordinates the adaptation of the `scope`, such as `price` in this example.

We note that the above adaptation rule changes the choreography and, as a consequence, the behavior of two of its roles. In general, an adaptation rule can impact an arbitrary number of roles. We also note that the need for adaptation is checked — and adaptation is possibly performed — every time the `scope` is executed. In this example, if the `client` buys many products, some with price above the threshold and some below, the need for adaptation is checked for each item and adaptation is performed only for the ones with a price above the threshold. In essence, purchases of cheap products follow the basic protocol whilst purchases of expensive ones follow the refined protocol introduced by the adaptation rule.

```
1  rule {
2   include log from "socket://localhost:8002"
3   include getTime from "socket://localhost:8003"
4   newRoles: logger
5   location@logger: "socket://localhost:15000"
6   on { N.scopename == "success-notification" }
7   do {
8    r@client = show( "Object delivered" )
9    |
10   {
11    log: seller( sel_product + " " + payment ) -> logger( entry );
12    time@logger = getTime();
13    log_entry = time + ": " + entry;
14    { r1@logger = log( log_entry ) | r2@logger = show( log_entry ) }
15   }
16  }
17 }
```

Listing 7.7 Adaptation rule: logging.

We now consider another need that may emerge. Assume that the `seller` decides to log all its sales, e.g., for tax payment reasons. Again, one may write an adaptation rule (see Listing 7.7) to answer this need. This rule targets the `scope` with property `N.scopename = "success-notification"` (lines 30–32 in Listing 7.5), which was not exactly intended for logging, but can be adapted to do so by taking care of repeating in the new code also the original notification message (line 31 in Listing 7.5, repeated at line 8 in Listing 7.7). The rule exploits both a new role, `logger`, and two external services `log` and `getTime`. External services are declared exactly as in AIOC programs. Note that here we omit the application-layer protocol of both services, hence the default, SOAP, is used.

The additional role is declared using keyword `newRoles` (line 4). New roles in AIOCJ rules should not be involved in the target AIOC program and take part to the choreography only while the body of the rule executes. As for normal roles, the URI of new roles is declared using the keyword `location`.

6 Deployment and Adaptation Procedure

In this section we describe the steps that DevOps need to follow to deploy the AIOCJ application of Listing 7.5 and to adapt it at runtime. When reporting paths, we use the Unix forward slash notation.

Compiling and Running an AIOC. As already mentioned, AIOCJ IDE runs as an Eclipse plugin. Hence, to create a new AIOC program we create a new project and a new file inside it with `.ioc` extension. We write the code in Listing 7.5 and we compile it by clicking on the button "Jolie Endpoint Projection"  . The compilation creates three folders in the Eclipse project: `epp_aioc`, `adaptation_manager`, and `environment`.

Within the folder `epp_aioc` we can find one subfolder for each role in the AIOC program containing all the related code. The main file is named after the role and has the standard Jolie extension `.ol`. The subfolder needs to be moved in the host corresponding to the location of the role declared in the `preamble` of the AIOC program. For example, the subfolder `client` should be moved into the host located at `"client.com"`.

Within the folders `adaptation_manager` and `environment` the main files are, respectively, `main_adaptationManager.ol` and `environment.ol`.

Before starting the compiled AIOC program, we make sure that the external services included in the choreography are running. To run the AIOC program, we first launch the Adaptation Manager with

```
jolie adaptation_manager/main_adaptationManager.ol
```

Then, we run the roles in the choreography, beginning from the `client`, which is declared as the `starter` of the choreography. For instance, the `client` — previously deployed at `"client.com"` — can be launched with

```
jolie client/client.ol
```

At the moment there is no need to run the Environment. As soon as the last role is started, the execution of the AIOCJ application begins.

Adapting a Running AIOC. Adaptation rules are defined using the same Eclipse plugin as AIOC programs. They need to be stored in a new `.ioc` file, either in the same project as the AIOC program or in a new one.

As for AIOC programs, the compilation of a set of adaptation rules is triggered by the "Jolie Endpoint Projection" button  and produces a folder named `epp_rules`, which corresponds to a unique Adaptation Server. Inside the folder, the main file is `AdaptationServer.ol` within path

```
__adaptation_server/servers/server
```

Also in this case, before starting the Adaptation Server, we make sure that the external services included in the rules are running.

If some adaptation rule has an applicability condition that checks some Environment variables (e.g., variable `E.split_payment_threshold` in Listing 7.6, line 3), the Environment microservice needs to be launched, running the program `environment.ol`. Environment variables can be added and removed both by console interaction or by editing the configuration file `environmentVariables.xml`.

If some adaptation rule needs a new role, the `location` declared for it should be able to interact with the Adaptation Server that contains the rule. To this end, AIOCJ provides a dedicated microservice called Role Supporter, which needs to be deployed in the host corresponding to the target `location`. This is done by moving to the corresponding host the folder

$$\texttt{role\_supporter/rule}N/roleName$$

where N is the sequential number of the rule, from top to bottom, inside the file `.ioc`, and $roleName$ is the name of the new role. The folder contains the code of the utility microservice, `RoleSupporter.ol`, and an automatically generated configuration file `config/location.iol`. For instance, the configuration file for the RoleSupporter for role `logger` in the rule in Listing 7.7 (assuming it is the only rule in the `.ioc` file) is

```
role_supporter/rule1/logger/config/locations.iol
```

If the location of the new role is unspecified, then `"localhost:9059"` is used by default and the corresponding folder is `default_role_supporter`.

Once both the external services and the Role Supporters are running, we can launch the Adaptation Server. When launched, the Adaptation Server registers at the Adaptation Manager and the compiled adaptation rules become enabled. From now on, when a `scope` is reached during execution, the rules in the Adaptation Server are checked for applicability.

Both microservices implementing roles of AIOCJ applications and the ones in AIOCJ RE — namely the Adaptation Manager, the Environment, the Adaptation Servers, and the Role Supporters — can be re-deployed on hosts different from the default ones. This requires to move the corresponding folder, but also to update the configuration files that contain their addresses, including their own configuration file. Notably, no recompilation is needed. We report in Figure 3 the dependency graph among the locations of AIOCJ microservices. In the figure, the notation **A** $\rightarrow$ **B** means that microservice **A** must know the deployment location of microservice **B**. At the bottom of each box we

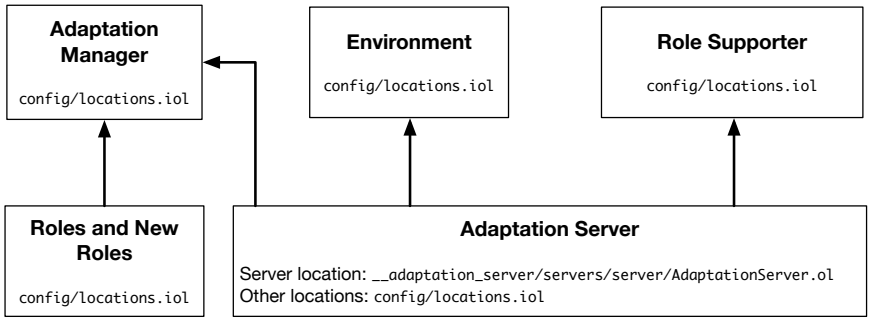

Figure 3 Location dependency graph among AIOCJ microservices.

report the path to the corresponding configuration file for locations, which is `config/locations.iol` except for the deployment location of Adaptation Servers which is directly contained in their own main file.

7 Conclusion

In this tutorial we have given a gentle introduction to the AIOCJ framework and to the AIOC language. While both adaptation and choreographies are thoroughly studied in the literature, their combination has not yet been explored in depth. As far as we know, AIOCJ is the only implemented framework in this setting. Theoretical investigations of the interplay between adaptation and multiparty session types [14, 15, 16] (which use choreographies as types instead of as a language) have been undertaken. A relevant work considers self-adaptive systems [14]. It uses multiparty session types to monitor that the computation follows the expected patterns, and it performs adaptation by moving from one choreography to the other according to external conditions. However, all possible behaviors are present in the system since the very beginning. Another work studies how to update a system so to preserve the guarantees provided by multiparty session types [15]. Another study, still preliminary, describes multiparty session types that can be updated from both inside and outside the system [16]. None of the three proposals above has been implemented. On the other side, we find two implemented approaches for programming using choreographies, Scribble [4, 17] and Chor [2], but they do not support adaptation. Chor is particularly related to AIOCJ, since they both produce Jolie code and they share part of the codebase. Finally, the main standard in the field of choreographic specifications, WS-CDL [5], does

not support adaptation. Moreover, WS-CDL is just a specification language and not an executable one. Further information on choreographies can be found in two surveys. One presents a general description of the theory of choreographies and session types [18]. The other accounts for their use in programming languages [19].

As future work we would like to understand what is needed to make AIOCJ more usable in practice. To this end, we are experimenting by applying AIOCJ to case studies developed for other approaches to adaptation, such as Context-Oriented Programming [20] and distributed [21] and dynamic [22] Aspect-Oriented Programming. Initial results in this direction can be found on the AIOCJ website [11]. Another direction is to provide automated support for the deployment of AIOCJ applications using containerization technologies such as Docker [23].

References

[1] M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centered programming for web services," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 2, 2012.

[2] M. Carbone and F. Montesi, "Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming," in *POPL*, pp. 263–274, ACM, 2013.

[3] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro, "Bridging the Gap between Interaction- and Process-Oriented Choreographies," in *SEFM*, pp. 323–332, IEEE, 2008.

[4] K. Honda, A. Mukhamedov, G. Brown, T. Chen, and N. Yoshida, "Scribbling interactions with a formal foundation," in *ICDCIT*, vol. 6536 of *LNCS*, pp. 55–75, Springer, 2011.

[5] World Wide Web Consortium, *Web Services Choreography Description Language Version 1.0*, 2005. http://www.w3.org/TR/ws-cdl-10/.

[6] F. Montesi, "Kickstarting choreographic programming," in *WS-FM:FASOCC*, vol. 9421 of *LNCS*, pp. 3–10, Springer, 2014.

[7] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, pp. 329–339, ACM, 2008.

[8] S. Newman, *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[9] M. Dalla Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro, "Dynamic choreographies - safe runtime updates of distributed applications," in *COORDINATION*, vol. 9037 of *LNCS*, pp. 67–82, Springer, 2015.

[10] M. Dalla Preda, S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli, "AIOCJ: A choreographic framework for safe adaptive distributed applications," in *SLE*, vol. 8706 of *LNCS*, pp. 161–170, Springer, 2014.

[11] "AIOCJ website." http://www.cs.unibo.it/projects/jolie/aiocj.html.

[12] "Jolie website." http://www.jolie-lang.org/.

[13] F. Montesi, C. Guidi, and G. Zavattaro, "Composing services with JOLIE," in *Proc. of ECOWS'07*, pp. 13–22, IEEE, 2007.

[14] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri, "Self-adaptive multiparty sessions," *Service Oriented Computing and Applications*, vol. 9, no. 3-4, pp. 249–268, 2015.
[15] G. Anderson and J. Rathke, "Dynamic software update for message passing programs," in *APLAS*, vol. 7705 of *LNCS*, pp. 207–222, Springer, 2012.
[16] M. Bravetti *et al.*, "Towards global and local types for adaptation," in *SEFM Workshops*, vol. 8368 of *LNCS*, pp. 3–14, Springer, 2013.
[17] "Scribble website." http://www.jboss.org/scribble.
[18] H. Hüttel *et al.*, "Foundations of session types and behavioural contracts," *ACM Computing Surveys*, vol. 49, no. 1, 2016.
[19] D. Ancona *et al.*, "Behavioral types in programming languages," *Foundations and Trends in Programming Languages*, vol. 3, no. 2-3, pp. 95–230, 2016.
[20] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented Programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, 2008.
[21] R. Pawlak *et al.*, "JAC: an aspect-based distributed dynamic framework," *Software: Practice and Experience*, vol. 34, no. 12, pp. 1119–1148, 2004.
[22] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley, "An aspect-oriented approach to dynamic adaptation," in *WOSS*, pp. 85–92, ACM, 2002.
[23] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2014.