The Evolution of Jolie

Ivan Lanese, Fabrizio Montesi, Gianluigi Zavattaro

▶ To cite this version:

Ivan Lanese, Fabrizio Montesi, Gianluigi Zavattaro. The Evolution of Jolie: From Orchestrations to Adaptable Choreographies. Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering, Mar 2015, Munich, Germany. pp.506-521, 10.1007/978-3-319-15545-6_29 . hal-01227623

HAL Id: hal-01227623

https://hal.inria.fr/hal-01227623

Submitted on 11 Nov 2015

The Evolution of Jolie
From Orchestrations to Adaptable Choreographies

Ivan Lanese[1], Fabrizio Montesi[2], and Gianluigi Zavattaro[1]

[1] Dep. of Computer Science and Engineering, INRIA FoCUS Team - Univ. Bologna
Mura A. Zamboni 7, 40127 Bologna - Italy
{lanese,zavattar}@cs.unibo.it
[2] Dep. of Mathematics and Computer Science, University of Southern Denmark
Campusvej 55, 5230 Odense - Denmark
fmontesi@imada.sdu.dk

Abstract. Jolie is an orchestration language conceived during Sensoria, an FP7 European project led by Martin Wirsing in the time frame 2005–2010. Jolie was designed having in mind both the novel –at project time– concepts related to Service-Oriented Computing and the traditional approach to the modelling of concurrency typical of process calculi. The foundational work done around Jolie during Sensoria has subsequently produced many concrete results. In this paper we focus on two distinct advancements, one aiming at the development of dynamically adaptable orchestrated systems and one focusing on global choreographic specifications. These works, more recently, contributed to the realisation of a framework for programming dynamically evolvable distributed Service-Oriented applications that are correct-by-construction.

1 Introduction

Sensoria (Software Engineering for Service-Oriented Overlay Computers)[3] is a research project funded by the European Commission under the 7-th Framework Programme in the time frame 2005–2010. Supervised by Martin Wirsing, the project has defined a novel approach for the engineering of advanced Service-Oriented applications. In fact, at project time, Service-Oriented Computing (SOC) was emerging as a novel and promising technology but, as frequently happens, the success of a promising technology requires the establishment of a mature methodology for the development of applications based on such technology. Easy-to-use tools are also needed to make the development methodology popular and largely diffused.

The success of Sensoria is demonstrated by the realisation of an integrated set of theoretical and concrete tools. In particular, UML-like visual languages have been developed for the high-level modelling of Service-Oriented applications (see, e.g., [18,7]), several process calculi have been designed to formally represent the operational aspects of such applications (see, e.g., [4,30,29,5]), analysis

[3] http://www.sensoria-ist.eu

techniques have been developed to perform both qualitative and quantitative analysis on these formal models of the applications (see, e.g., [19,37,13]), and also runtime support for application deployment has been realised by providing some of the proposed calculi with an execution environment (see, e.g., [34,31]).

Among the implementations of process calculi, Jolie [34] was conceived as a fully fledged Service-Oriented programming language that, on the one hand, is based on a formally defined semantics and, on the other hand, turns out to be easy to integrate with state-of-the-art technologies (in particular those based on the Java platform). This marked technological flavour of the language is witnessed by the birth of the spin-off company *italianaSoftware s.r.l.* [4] whose mission is indeed the exploitation in industrial environments of Jolie and its related tools.

Beyond the initial development of Jolie within Sensoria, many advancements took place even after Sensoria ended. For instance, in [22] Jolie is extended with primitives for fault handling and compensations; in [15] linguistic primitives are proposed for easy realisation of software architectures including advanced connectors like proxies, wrappers, and load balancers; and in [21] Jolie is studied as a language for workflow management by showing how to implement the popular van der Aalst's workflow patterns [38].

In this paper, we narrate some of the extensions of Jolie in order to demonstrate how our experience in the development of an orchestration language during Sensoria provided us with valuable insights in related research lines.

We start by presenting two independent frameworks based on Jolie. The first one is JoRBA [27], a framework for the programming of service-oriented applications that can dynamically adapt to their execution context. JoRBA uses the code mobility mechanisms offered by Jolie to adapt the behaviour of services at runtime; the need for adaptation is decided by special rules given by programmers in a declarative way. The second one is Chor [11], a choreographic language allowing for the programming of distributed systems from a global viewpoint. Chor is equipped with a compiler that generates executable Jolie programs, which are guaranteed to be correct by construction from the originating global program. We finally present AIOCJ [16], a choreography language that combines the global approach of Chor to distributed programming with the adaptability features of JoRBA. The Jolie programs compiled by AIOCJ are guaranteed to be correct not only as far as the initial code is concerned, but also when the updates specified in AIOCJ are dynamically applied to the application endpoints.

Paper structure. In Section 2 we recall the basics of the Jolie language. In Section 3 we discuss the importance of dynamic adaptation for modern applications and present JoRBA, an extension of Jolie that supports rule-based dynamic program updates. In Section 4 we discuss Chor, the Jolie based approach to the correct-by-construction realisation of communication-centred distributed applications. Finally, in Section 5 we present how in AIOCJ we have been able to combine the rule-based dynamic adaptation mechanisms explored with JoRBA

[4] `http://www.italianasoftware.com`

to the correct-by-construction approach characterising Chor. Related work is discussed in Section 6. We report some conclusive remarks in Section 7.

2 Service-Oriented Programming with Jolie

Jolie programs contain two parts related, respectively, to the *behaviour* of a service and to its *deployment*. Here, we describe in more details the basic primitives for programming the behavioural part of Jolie, while we simply mention that the deployment part is used to instantiate networking and lower-level communication aspects like the definition of the ports and of the communication protocols to be used.

Jolie combines message-passing and imperative programming style. Scopes, indicated by curly brackets `{...}`, are used to represent blocks and procedures. Procedures are labelled with the keyword `define`; the name of a procedure is unique within a service and the `main` procedure is the entry point of execution for each service. Traditional sequential programming constructs like conditions, loops, and sequence are standard. In addition, Jolie includes a *parallel* operator `|` that executes the left and right activities concurrently. Concerning communication, Jolie supports two kinds of message-passing operations, namely *One-Way*s (OWs) and *Request-Response*s (RRs). On the sender's side, the former operation sends a message and immediately passes the thread of control to the subsequent activity in the process, while the latter sends a request and keeps the thread of control until it receives a response. On the receiver's side, OWs receive a message and store it into a defined variable, whilst RRs receive a message, execute some internal code, and finally send the content of a second variable as response.

Jolie provides also an input-guarded choice $[\eta_1]\{B_1\}\dots[\eta_n]\{B_n\}$, where η_i, $i \in \{1,\dots,n\}$, is an input statement and B_i is the behaviour of the related branch. When a message on η_i is received, η_i is executed and all other branches are deactivated. Afterwards, B_i is executed. A static check enforces all the input choices to have different operations to avoid ambiguity.

Figure 1 reports a Jolie program taken from [21] including two services `A` and `B`. `A` sends in parallel the content of variables `a` and `b` through OW operations `op1` and `op2`, respectively, at (`@`) `B`. When `B` receives a message on one of the corresponding OW operations, it stores the content of the message in the corresponding variable. After the completion of the scope at Lines 2-5, `A` proceeds with the subsequent RR operation `op3`, which sends the content of variable `e` and stores its response in `h`. The scope linked to `op3`, in Lines 6-8 of service `B`, is the activity executed before sending the response to `A`; as this activity assigns `"Hello, world"` to the return variable `g`, this string is returned to `A`, and thus `A` assigns it to its return variable `h`. The command at Line 8 of the service `A` sends the content of `h` to the `println` operation of the `Console`; in this way `"Hello, world"` is printed.

An interesting feature of Jolie is that it provides *dynamic embedding*. Dynamic embedding is a mechanism allowing to take the code of a Jolie service and dynamically run the service inside the current application. This mechanism

```
//service A
{
  op1@B( a )
  | op2@B( b )
};
op3@B( e )( h );
println@Console( h )()
```

```
//service B
{
  op1( c )
  | op2( d )
};
op3( f )( g ){
  g = "Hello, world"
}
```

Fig. 1. An example of composition and communication between services.

is used quite extensively when programming adaptive applications in Jolie, since it allows one to dynamically load new code to deal with unexpected needs.

3 Managing Dynamic Adaptation with JoRBA

Modern software applications change their behaviour, reconfigure their structure and evolve over time reacting to changes in the operating conditions, so to always meet users' expectations. This is fundamental since those applications live in distributed and mobile devices, such as mobile phones, PDAs, laptops, etc., thus their environment may change frequently. Also, user goals and needs may change dynamically, and applications should adapt accordingly, without intervention from technicians. We aim at *dynamic adaptation*, where the application is able to face unexpected adaptation needs. Dynamic adaptation is challenging since information on the update to be performed is not known when the application is designed, deployed, or even started.

JoRBA (Jolie Rule-Based Adaptation framework) [27] is a Jolie-based framework for programming adaptable applications, which is based on the separation between the application behaviour and the adaptation specification. An adaptable application should provide some *adaptation hooks*, i.e., information on part of its structure and its behaviour. The adaptation logic should be developed separately, for instance as a set of adaptation rules, by some adaptation engineer, and can be created/changed after the application has been deployed without affecting the running application. Adaptation should be enacted by an *adaptation middleware*, including an *adaptation manager* and some, possibly distributed, *adaptation servers*. The latter are services that act as repositories of adaptation rules. Running adaptation servers register themselves on the adaptation manager. The running application may interact with the adaptation manager to look for applicable adaptation rules. Whether a rule is applicable or not may depend on environment conditions (e.g., date, workload), including user preferences, and on properties of the current implementation (e.g., performance, code version, cost). The adaptation manager checks the available rules and returns one of them which can be applied, if it exists. The effect of an adaptation rule is to replace an activity with new code that answers a given adaptation need.

Activity	**Functional Parameters**			**Non-functional Parameters**	
`Activity Name`	`Number`	`Source`	`Destination`	`Time`	`Cost`
Take Train	IC2356	Bologna Train Station	Munich Train Station	7 h 41 m	80 euros
Take Bus	13	Munich Train Station	LMU	30 m	1 euro
Take Taxi	25	Munich Train Station	LMU	10 m	15 euros
Go To Meeting	-	Bob's House	LMU	9 h	100 euros

Table 1. List of possible (Travelling) domain activities.

We describe now on a sample scenario the approach used in JoRBA to deal with dynamic adaptation.

Travelling Scenario Consider Bob travelling from Bologna to LMU (Martin Wirsing's university) for a Sensoria meeting. He may have on his mobile phone an application instructing him about what to do, taking care of the related tasks. A set of possible tasks are in Table 1. For instance, the activity *Take Train* connects to the information system of Bologna train station to buy the train ticket. It also instructs Bob about how to take the train, and which one to take.

Assume that such an application has been developed for adaptation. This means that its *adaptation interface* specifies that some of the activities are adaptable. Each adaptable activity has a few parameters, e.g., *Number*, specifying the code of the train, bus or taxi to be taken, *Source* specifying the desired leaving place and *Destination* specifying the desired arrival place, all visible from the adaptation interface. Also, a few non-functional parameters for the activities may be specified as *Time* and *Cost*. We show now a couple of examples of adaptation.

Example 1. When Bob arrives to Bologna train station, its Travelling application connects to the adaptation server of the train station. Assume that a new "*Italo*"(Italian high speed train) connection has been activated from Bologna to Munich providing a connection with *Time=4 h 23 m* and *Cost=92 euros*. This is reflected by the existence of an adaptation rule specifying that all the applications providing an activity *Take Train* for a train for which the new connection is possible may be adapted. Adaptation may depend on Bob's preferences for comparing the old activity and the new one, or may be forced if, for instance, the old connection is no more available. If adaptation has to be performed, the new code for the activity is sent by the adaptation server to the application, where it becomes the new definition of activity *Take Train*. Note that the new code can be quite different from the old one, e.g., if the new trains are booked using a different communication protocol. Thus Bob can immediately exploit the new high speed connection, which was not expected when the application has been created.

Example 2. Suppose that the train from Bologna to Munich is one hour late. Bob mobile phone may have an adaptation server taking care of adapting all Bob's applications to changing environment conditions. The adaptation server will be notified about the train being late, and it may include an adaptation rule specifying that if Bob is late on his travel, he can take a taxi instead of arriving to LMU by bus. The adaptation rule thus replaces the activity *Take Bus* of the travelling application with a new activity *Take Taxi*. Again, this can be done even if different protocols and servers are used to buy bus tickets and to reserve a taxi.

A Rule-based Approach to Dynamic Adaptation Instead of presenting the syntax of JoRBA, we discuss its approach to dynamic adaptation which is general enough to be applied to applications developed using any other language, provided that (i) the application exposes the desired adaptation interface and (ii) the language is able to support the code mobility mechanism necessary for performing adaptation. At the end of this section we briefly show that Jolie supports both these features.

Thus we want to build an adaptable application using some language L and following the approach above to dynamic adaptation. The application must expose a set of *adaptable domain activities* (or, simply, activities) $\{A_i\}_{i \in I}$, together with some additional information. Activities A_i are the ones that may require to be updated to adapt the application to changes in the operating conditions. While it is necessary to guess where adaptation may be possible, it is not necessary to know at the application development time which actual conditions will trigger the adaptation, and which kind of adaptation should be performed.

The adaptable application will interact with an adaptation middleware providing the adaptation rules. The environment has full control over the set of rules, and may change them at any time, regardless of the state of the running application. Each such rule includes a description of the activity to be adapted, an applicability condition specifying when the rule is applicable, the new code of the activity, the set of variables required by the activity, and some information on the non-functional properties of the activity.

At runtime, the rule is matched against the application activity to find out whether adaptation is possible/required. In particular:

- the description of the activity to be adapted in the rule should be compatible with the description of the activity in the application;
- the applicability condition should evaluate to true; the applicability condition may refer to both variables of the environment (retrieved by the adaptation manager) and variables published by the adaptation interface of the application;
- the non-functional properties guaranteed by the new code provided by the adaptation rule should be better than the ones guaranteed by the old implementation, according to some user specified policy;
- the variables required by the new code should be a subset of the variables provided by the application for the activity.

If all these conditions are satisfied then adaptation can be performed, i.e. the new code of the activity should be sent by the adaptation manager to the application, and installed by the application replacing the old one. Since the update may also influence the state, we also allow the adaptation rule to specify a state update for the adaptable application.

More precisely, the following steps are executed:

1. the adaptation server sends the new code to the application, which replaces the old code of the activity;
2. the adaptation interface of the application is updated, with the new non-functional properties, e.g., *Time=4 h 23 m* and *Cost=92 euros*, replacing the old ones;
3. the state of the application is updated, e.g., by setting variable $Number$ to $IT82$, the number of the "*Italo*" train.

The first step is the most tricky, since the new code needs to be sent from the adaptation server to the application and integrated with the rest of the application. For instance, it should be able to exploit the public variables of the application.

JoRBA is a proof-of-concept implementation of our adaptation mechanism based upon the Jolie language. Indeed, both the adaptation middleware, including the distributed adaptation servers, and the adaptable application are built as Jolie services. Thus, interactions between them are obtained using Jolie OWs and RRs communication primitives. The code inside adaptable activities is externalized from the main body of the application as a separate service. In this way adaptation is realized by disabling the service implementing the adaptable activity and replacing it with the new code coming from the adaptation manager, which is launched using Jolie dynamic embedding. Since both the main part of the application, the old service and the new one should share part of the state, this is externalized as a separate service accessible from all of them.

4 Correct-by-construction Development with Chor

In the context of Service-Oriented Computing and, more in general, for the development of distributed communication-centred applications, the top-down approach based on *global* specifications that are automatically projected to *end-point* code has recently emerged as a popular approach for the realisation of correct-by-construction applications [10,25,40,28,12]. Global specifications are expressed using so-called *choreography* languages: the message flow among the partners in the application is expressed from a global viewpoint as it happens, e.g., in Message Sequence Charts [23] or when security protocols are specified by using actions like, e.g., Alice $\rightarrow$ Bob : $\{M\}_k$ meaning that Alice sends to Bob the message M encrypted with the key k. These global specifications are guaranteed to be correct, in particular *deadlock- and race-free*, because only successful and completed communications among two interacting partners can be expressed. In other terms, it is not possible to specify an endpoint that remains blocked

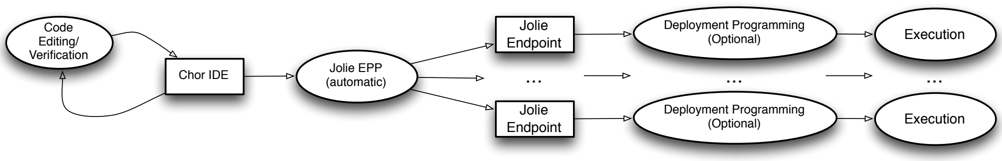
Code Editing/ Verification
Chor IDE
Jolie EPP (automatic)
Jolie Endpoint
...
Jolie Endpoint
Deployment Programming (Optional)
...
Deployment Programming (Optional)
Execution
...
Execution

Fig. 2. Chor development methodology, from [11].

waiting indefinitely for a never arriving message. The actual communication behaviour of each single partner is in turn obtained by projection from the global specification: the obtained projected code is guaranteed to adhere to the global specification, thus correctness is preserved.

This popular approach has been also adapted to Jolie. In this case, the choreographic language is Chor [11]. Chor offers a programming language, based on choreographies, and an Integrated Development Environment (IDE) developed as an Eclipse plugin for the writing of programs. In the development methodology suggested with Chor, depicted in Figure 2, developers can first use the IDE to write protocol specifications and choreographies. The programmer is supported by on-the-fly verification which takes care of checking (i) the syntactic correctness of program terms and (ii) the type compliance of the choreography w.r.t. the protocol specifications, using a (behavioural) typing discipline.

Once the global program is completed, developers can automatically project it to an endpoint implementation. Endpoint implementations are given in Jolie. Nevertheless, Chor is designed to be extended to multiple endpoint languages: potentially, each process in a choreography could be implemented with a different endpoint technology.

Each Jolie endpoint program comes with its own deployment information, given as a term separated from the code implementing the behaviour of the projected process. This part can be optionally customised by the programmer to adapt to a specific communication technology. Finally, the Jolie endpoint programs can be executed; as expected, they will implement the originating choreography.

In order to give an idea of how global specifications can be expressed in Chor, we present a simple example.

Example 3 (Chor program example).

```
program simple;

protocol SimpleProtocol { C -> S: hi( string ) }

public a: SimpleProtocol

main
{
 9   client[C] start server[S] : a( k );
10   ask@client( "[client] Message?", msg );
11   client.msg -> server.x : hi( k );
12   show@server( "[server] " + x )
13 }
```

Program `simple` above starts by declaring a protocol `SimpleProtocol`, in which role `C` (for client) sends a string to a role `S` (for server) through operation `hi`. In the choreography of the program, process `client` and a fresh service process `server` start a session `k` by synchronising on the public channel `a`.[5] Process `client` then asks the user for an input message and stores it in its local variable `msg`, which is then sent to process `server` through operation `hi` on session `k`, implementing protocol `SimpleProtocol`. Finally, process `server` displays the received message on screen.

As mentioned above, Chor has been equipped with an automatic endpoint projection (EPP) that generates Jolie code; the following example shows (part of) the result of the endpoint projection of the simple choreography presented above.

Example 4 (Endpoint Projection in Chor). We give an example of EPP by reporting a snippet of the code generated for process `server` from Example 3:

```
 1 main
 2 {
 3   _start();
 4   csets.tid = new;
 5   _myRef.binding << global.inputPorts.MyInputPort;
 6   _myRef.tid = csets.tid;
 7   _start_S@a(_myRef)(_sessionDescriptor.k);
 8   k_C << _sessionDescriptor.k.C.binding;
 9   hi(x);
10   showMessageDialog@SwingUI("[server] " + x)()
11 }
```

The Jolie code above for process `server` waits to be started by receiving an input on operation `_start`. This starts the generation of the session `k` (Lines 4–8, that we do not comment in detail). Finally, in Lines 9–10, the `server` receives the message on operation `hi` from the client and displays it on screen as indicated by the choreography.

5 Correct-by-construction Adaptive Applications

The JoRBA approach to adaptation and the use of choreographies to ensure that applications are deadlock- and race-free by construction can be combined. We

[5] Session keys are necessary to keep track of the protocols: see, e.g., the presence of the session key `k` in Line 11 indicating that this interaction between `client` and `server` is part of the protocol started at Line 9.

describe below AIOCJ [16][6], a framework to program adaptive choreographies. AIOCJ combines an Eclipse plugin to edit adaptable applications and generate code for each participant using a projection similar to the one of Chor, with an adaptation middleware similar to the one of JoRBA managing adaptation. The main point is that adaptation should be coordinated, so to ensure that no error occurs because of inconsistent updates.

We consider applications composed by processes deployed as services on different localities, including local state and computational resources. Each process has a specific duty in the choreography. As for JoRBA, adaptation is performed by interacting with an adaptation middleware storing adaptation rules. The main difference is that now a rule requires to update many participants of the choreography in a coordinated way. The parts of the choreography to be updated are syntactically delimited by *adaptation scopes*.

The language for programming AIOCJ applications relies on a set of roles that identify the processes in the choreography. Let us introduce the syntax of the language using an example where Bob invites Alice to see a film (Listing 1.1).

The code starts with some deployment information (Lines 1-9) that we discuss later on. The description of the behaviour starts at Line 11. The program is made by a loop where Bob first checks when Alice is available and then invites her to the cinema. Before starting the loop, Bob initialises the variable `end` to the boolean value **`false`** (Line 12). The variable is used to control the exit from the loop. Note the annotation `@bob` meaning that `end` is a local variable of Bob. The first instructions of the while loop are enclosed in an adaptation scope (Lines 14-18), meaning that this part of the program may be adapted in the future. The first operation within the adaptation scope is the call to the primitive function `getInput` that asks to Bob a day where he is free and stores this date into the local variable `free_day`. At Line 16 the content of `free_day` is sent to Alice via operation `proposal`. Alice stores it in its local variable `bob_free_day`. Then, at Line 17, Alice calls the external function `isFreeDay` that checks whether she is available on `bob_free_day`. If she is available (Line 19) then Bob sends to her the invitation to go to the cinema via the operation `proposal` (Line 21). Alice, reading from the input, accepts or refuses the invitation (Line 25). If Alice accepts the invitation then Bob first sets the variable `end` to **`true`** to end the loop. Then, he sends to the cinema the booking request via operation `book`. The cinema generates the tickets using the external function `getTicket` and sends them to Alice and Bob via operation `notify`. The two notifications are done in parallel using the parallel operator `|` (until now we composed statements using the sequential operator `;`). Lines 20-32 are enclosed in a second adaptation scope with property `N.scope_name = "event selection"`. If the agreement is not reached, Bob decides, reading from the input, if he wants to stop inviting Alice. If so, the program exits setting the variable `end` to **`true`**.

We remark the different possible meanings of annotations such as `@bob` and `@alice`. When prefixed by a variable, they identify the owner of the variable. Prefixed by the boolean guard of conditionals and loops, they identify the role

[6] `http://www.cs.unibo.it/projects/jolie/aiocj.html`

```
1  include isFreeDay from "calendar.org:80" with http
2  include getTicket from "cinema.org:8000" with soap
3
4  preamble {
5    starter: bob
6    location@bob = "socket://localhost:8000"
7    location@alice = "socket://alice.com:8000"
8    location@cinema = "socket://cinema.org:8001"
9  }
10
11 aioc{
12   end@bob = false;
13   while( ! end )@bob{
14     scope @bob {
15       free_day@bob = getInput( "Insert your free day" );
16       proposal: bob( free_day ) -> alice( bob_free_day );
17       is_free@alice = isFreeDay( bob_free_day );
18     } prop { N.scope_name = "matching day" };
19     if( is_free )@alice {
20       scope @bob {
21         proposal: bob( "cinema" ) -> alice( event );
22         agreement@alice = getInput( "Bob proposes "  + event +
23           ", do you agree?[y/n]");
24         if( agreement == "y" )@alice{
25           end@bob = true;
26           book: bob( bob_free_day ) -> cinema( book_day );
27           ticket@cinema = getTicket( book_day );
28           { notify: cinema( ticket ) -> bob( ticket )
29             | notify: cinema( ticket ) -> alice( ticket )
30           }
31         }
32       } prop { N.scope_name = "event selection" }
33     };
34     if( !end )@bob {
35       _r@bob = getInput( "Alice refused. Try another date?[y/n]" );
36       if( _r != "y" )@bob{ end@bob = true }
37     }
38   }
39 }
```

Listing 1.1. Appointment program.

that evaluates the guard. Prefixed by the keyword **scope**, they identify the process coordinating the adaptation of that scope. An adaptation scope, besides the code, may also include some properties describing the current implementation. These can be specified using the keyword `prop` and are prefixed by `N`. For instance, each adaptation scope of the example includes the property `scope_name`, that can be used to find out its functionality.

AIOCJ can interact with external services, seen as functions. This allows both to interact with real services and to have easy access to libraries from other languages. To do that, one must specify the address and protocol used to interact with each service. For instance, the external function `isFreeDay` used in Line 17 is associated to the service deployed at the domain "calendar.org", reachable though port 80, and that uses http as serialisation protocol (Line 1). External functions are declared with the keyword **include**. To preserve deadlock freedom, external services must be non-blocking. After function declaration, in a `preamble` section, it is possible to declare the locations where processes are deployed. The keyword `starter` is mandatory and defines which process must be started first. The starter makes sure all other processes are ready before the execution of the choreography begins.

Now suppose that Bob, during summer, prefers to invite Alice to a picnic rather than to the cinema, provided that the weather forecasts are good. This can be obtained by adding the following adaptation rule to one of the adaptation servers. This may even be done while the application is running, e.g., while Bob is sending an invitation. In this case, if Bob first try is unsuccessful, in the second try he will propose a picnic.

```
 1  rule {
 2    include getWeather from "socket://localhost:8002"
 3    on { N.scope_name == "event selection" and E.month > 5 and E.month < 10 }
 4    do {
 5      forecasts@bob = getWeather( free_day );
 6      if( forecasts == "Clear" )@bob{
 7        eventProposal: bob( "picnic" ) -> alice( event )
 8      } else { eventProposal: bob( "cinema" ) -> alice( event ) };
 9      agreement@alice = getInput( "Bob proposes "  + event +
10        ", do you agree?[y/n]");
11      if( agreement == "y" )@alice {
12        end@bob = true |
13        if( event == "cinema" )@alice {
14          //cinema tickets purchase procedure
15        }
16      }
17    }
18  }
```

Listing 1.2. Event selection adaptation rule.

A rule specifies its applicability condition and the new code to execute. In general, the applicability condition may depend only on properties of the adaptation scope, environment variables, and variables belonging to the coordinator of the adaptation scope. In this case, the condition, introduced by the keyword `on` (Line 3), makes the rule applicable to adaptation scopes having the property `scope_name` equal to the string `"event selection"` and only during summer. This last check relies on an environment variable `month` that contains the current month. Environment variables are prefixed by `E`.

The new code to execute if the rule is applied is defined using the keyword `do` (Line 4). The forecasts can be retrieved calling an external function `getWeather` (Line 5) that queries a weather forecasts service. This function is declared in Line 2. If the weather is clear, Bob proposes to Alice a picnic, otherwise he proposes the cinema. Booking (as in Listing 1.1, Lines 26-29) is needed only if Alice accepts the cinema proposal.

6 Related Work

Choreography-like methods for programming distributed systems have been applied for a long time, for example in MSC [26], security protocols [6,9,3] and automata theory [20]. Differently from Chor, these works were not intended as fully-fledged programming languages. For example, they do not deal with concrete data or different layers of abstraction (protocols and choreographies).

The development of Chor was partially inspired by the language WS-CDL [39] and the choreography fragment of BPMN [8]. Differently from those, Chor comes with a formal model defining its semantics and typing discipline. This model introduced a precise understanding of multiparty sessions and typical aspects of concurrency, such as asynchrony and parallelism, to choreographies [11]. The typing discipline is based on multiparty session types [25] (choreography-like protocols), bringing their benefits to choreographies; for example, Chor programs are statically guaranteed to follow their associated protocols (session fidelity, initially introduced in [24]). Remarkably, the development of Chor proved that the mixing of choreographies with multiparty session types yields more than just the sum of the parts. For example, it naturally supports a simple procedure for automatically inferring the protocols implemented by a choreography (type inference); and, it guarantees deadlock-freedom for a system even in the presence of arbitrary interleavings of session behaviours, without requiring additional machinery on top of types as is needed when dealing with processes instead of choreographies [2]. The theoretical model of Chor has been recently extended for supporting the reuse of external services in [35]. We refer the interested reader to [33] for a detailed explanation of these aspects and for an evaluation of Chor w.r.t. some concrete scenarios. Exploring a similar direction, Scribble is a choreography-like language for specifying communication protocols, based on multiparty session types [40]. Differently from Chor, Scribble protocols are not compiled to executable programs but to local abstract behaviours that are used to verify or monitor the concrete behaviour of endpoints in a distributed system (see, e.g., [36]).

Like choreographies, also adaptation is a lively research topic, as shown by the survey [32]. However, most of the approaches propose mechanisms for adaptation without any guarantee about the properties of the application after adaptation, as for JoRBA.

A few approaches try to apply multiparty session types to adaptation, obtaining some formal guarantee on the behaviour of the system. In this sense, they are different from Chor and AIOCJ, which are fully-fledged languages. For instance, [1] deals with dynamic software updates of systems which are concurrent, but not distributed. Furthermore, dynamic software updates are applied on demand, while enactment of adaptation depends on the environment and on the state of the running application. Another related approach is [14], which deals with monitoring of self-adaptive systems. There, all the possible behaviours are available since the very beginning, both at the level of types and of processes, and a fixed adaptation function is used to switch between them. This difference derives from the distinction between self-adaptive applications, as they discuss, and applications updated from the outside, as in our case. We also recall [17], which uses types to ensure no session is spoiled because of adaptation, and that needed services are not removed. However, [17] allows updates only when no session is active, while AIOCJ changes the behaviour of running interactions.

7 Conclusions

The aim of the European project Sensoria was to devise a methodology for the development of Service-Oriented applications, and realise the corresponding theoretical and practical tools. Most of the research effort of our research group in Bologna has been dedicated to the investigation of appropriate models and languages for specifying and programming such applications. The Service-Oriented programming language Jolie has been one of our main achievements. In this paper we have discussed how the activity initiated during the Sensoria project produced results far beyond our initial aims. In particular, we are still nowadays exploiting Jolie in the realisation of a framework for programming adaptable communication-centred applications that are correct-by-construction. Correctness is guaranteed because the updates are expressed from a global view-point, and then automatically projected and injected in the endpoints code. For instance, if in an application it is necessary to update a security protocol because of a detected flaw, the interacting partners need to be modified in order to replace the old protocol with a new one. Applying these dynamic unexpected updates is a critical task for modern applications. Our approach consists of describing the updates at the global level, generate the new endpoint code by projecting such updates on the affected partners, and then inject the new code to the endpoints in a coordinated manner.

A final remark is dedicated to Martin Wirsing, the coordinator of the Sensoria project. He did not limit his activity to the (hard) work of amalgamating the several heterogeneous partners of the Sensoria Integrated Project, but he continuously solicited the participants to conduct research that were innovative –in order to give to the project a long-term vision– as well as close to actual application needs –in order to avoid losing effort on abstract useless research. We consider the specific experience reported in this paper a concrete and relevant result of this enlightened Martin's approach to project coordination.

Acknowledgements

Montesi was supported by the Danish Council for Independent Research (Technology and Production), grant n. DFF–4005-00304.

References

1. G. Anderson and J. Rathke. Dynamic software update for message passing programs. In R. Jhala and A. Igarashi, editors, *APLAS*, volume 7705 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2012.
2. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2008.

3. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140. IEEE Computer Society, 2009.
4. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. SCC: A service centered calculus. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.
5. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In G. Barthe and F. S. de Boer, editors, *FMOODS*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2008.
6. S. Briais and U. Nestmann. A formal semantics for protocol narrations. *Theor. Comput. Sci.*, 389(3):484–511, 2007.
7. R. Bruni, M. M. Hölzl, N. Koch, A. Lluch-Lafuente, P. Mayer, U. Montanari, A. Schroeder, and M. Wirsing. A service-oriented UML profile with formal support. In L. Baresi, C.-H. Chi, and J. Suzuki, editors, *ICSOC/ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 2009.
8. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/`.
9. C. Caleiro, L. Viganò, and D. A. Basin. On the semantics of Alice&Bob specifications of security protocols. *Theor. Comput. Sci.*, 367(1-2):88–122, 2006.
10. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
11. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 263–274. ACM, 2013.
12. G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multiparty session. *Logical Methods in Computer Science*, 8(1), 2012.
13. A. Clark, S. Gilmore, and M. Tribastone. Quantitative analysis of web services using SRMC. In M. Bernardo, L. Padovani, and G. Zavattaro, editors, *SFM*, volume 5569 of *Lecture Notes in Computer Science*, pages 296–339. Springer, 2009.
14. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Self-adaptive monitors for multiparty sessions. In *PDP*, pages 688–696. IEEE Computer Society, 2014.
15. M. Dalla Preda, M. Gabbrielli, C. Guidi, J. Mauro, and F. Montesi. Interface-based service composition with aggregation. In F. D. Paoli, E. Pimentel, and G. Zavattaro, editors, *ESOCC*, volume 7592 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2012.
16. M. Dalla Preda, S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli. AIOCJ: A choreographic framework for safe adaptive distributed applications. In B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, editors, *SLE*, volume 8706 of *Lecture Notes in Computer Science*, pages 161–170. Springer, 2014.
17. C. Di Giusto and J. A. Pérez. Disciplined structured communications with consistent runtime adaptation. In S. Y. Shin and J. C. Maldonado, editors, *SAC*, pages 1913–1918. ACM, 2013.
18. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 2006.
19. H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 771–774. ACM, 2006.
20. X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *International Journal on Web Service Res.*, 2(4):68–93, 2005.

21. M. Gabbrielli, S. Giallorenzo, and F. Montesi. Service-oriented architectures: From design to production exploiting workflow patterns. In S. Omatu, H. Bersini, J. M. C. Rodríguez, S. Rodríguez, P. Pawlewski, and E. Bucciarelli, editors, *DCAI*, volume 290 of *Advances in Intelligent Systems and Computing*, pages 131–139. Springer, 2014.
22. C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. Dynamic error handling in service oriented applications. *Fundam. Inform.*, 95(1):73–102, 2009.
23. D. Harel and P. Thiagarajan. Message sequence charts. In *UML for real*, pages 77 – 105. Kluwer Academic Publishers, 2003.
24. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 22–138. Springer, 1998.
25. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.
26. International Telecommunication Union. Recommendation Z.120: Message sequence chart, 1996.
27. I. Lanese, A. Bucchiarone, and F. Montesi. A framework for rule-based dynamic adaptation. In M. Wirsing, M. Hofmann, and A. Rauschmayer, editors, *TGC*, volume 6084 of *Lecture Notes in Computer Science*, pages 284–300. Springer, 2010.
28. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*, pages 323–332. IEEE Computer Society, 2008.
29. I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *SEFM*, pages 305–314. IEEE Computer Society, 2007.
30. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In R. De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.
31. A. Lapadula, R. Pugliese, and F. Tiezzi. Using formal methods to develop WS-BPEL applications. *Sci. Comput. Program.*, 77(3):189–213, 2012.
32. L. A. F. Leite, G. A. Oliva, G. M. Nogueira, M. A. Gerosa, F. Kon, and D. S. Milojicic. A systematic literature review of service choreography adaptation. *Service Oriented Computing and Applications*, 7(3):199–216, 2013.
33. F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. `http://www.fabriziomontesi.com/files/m13_phdthesis.pdf`.
34. F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proc. of ECOWS*, pages 13–22. IEEE Computer Society, 2007.
35. F. Montesi and N. Yoshida. Compositional choreographies. In P. R. D'Argenio and H. C. Melgratti, editors, *CONCUR*, volume 8052 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2013.
36. R. Neykova, N. Yoshida, and R. Hu. SPY: local verification of global protocols. In A. Legay and S. Bensalem, editors, *RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 358–363. Springer, 2013.
37. M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In S. Leue and P. Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2007.
38. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
39. W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.

40. N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In M. Abadi and A. Lluch-Lafuente, editors, *TGC*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013.