Toward a Definition of π-DSL for Modelling Business Agents

MDA based π–calculus extension

Charif Mahmoudi and Fabrice Mourlin
Laboratory of Algorithms, Complexity and Logics,
Paris 12th University
Créteil, France
{charif.mahmoudi, fabrice.mourlin}@u-pec.fr

***Abstract*—In this paper, we will address the issue of modeling the integration of agents with various resources and services, as found in an Service-Oriented Architecture (SOA) platform. We are proposing an approach for modeling agents and integrating these agents in existing pipes and filters based message routing and mediation engines. Using Model-driven development (MDA) as a base for our modeling strategy, our agent model generates source code based on Enterprise integration patterns (EIP) by Hohpe and Woolf. We are presenting a new agent design that uses the Open Gateway Services Interfaces (OSGi) architecture as an agent platform and the Apache Camel enterprise integration framework as the EIP based engine. The approach is illustrated by a business process use case, and a complete example including process specification and code generation. The main objective of the example is to demonstrate the benefits of using agents as orchestration of external services via a specialized message routing engine that supports EIPs.**

Keywords- Process algebra; Orchestration languages; Software agents; Web services; EIP; π–DSL; MDA; SOA; OSGi

I. INTRODUCTION

In the business world, the orchestration of Web Services is becoming increasingly widespread [1] This technology allows, via tools, a simple way to handle graphically different business needs. We give as an example BPMN [2]. Other specifications can be described as the specification for the construction of orchestrations as Apache CAMEL [3] and Spring Integration [4]. For some researchers [5], the specifications based on based on Enterprise Integration Pattern (EIP) [6] are dedicated routing within ESB [7]. But most of them [8] agree that specifications based on EIP are ideal for building orchestrations. In addition, it should be noted that most of the specifications based on EIP do not offer graphical tools to develop visually unlike BPMN specification.

In this paper, we will present an approach allowing orchestrations in a mobile agent [9] form based on the EIP specifications. This approach is based on the work [10] that we previously published and which we consider as the foundation of an OSGi [11] based ecosystem able to run mobile agents.

The paper is organized as following. We review a number of related works in Section 2, and describe the standards we have set as a framework of our work in Section 3. Section 4 provides the detail of the MDA approach that we used to define our system. Section 5 presents the formal specifications of our EIP based target system. It uses EIP specifications as a mean to declare a mobile orchestration carrying agent [12]. We conclude our work, and describe the future work in Section 6.

II. WORK CONTEXT

In the context of SOA [13], the orchestration has a central role since it defines the steps to be performed to provide a result. The steps are Web services calls, the results of the various services are handled by the orchestrator. The final result of the orchestration is based on the results of each step.

The orchestrations are defined by the W3C (glossary) as "the pattern of interactions that must respect a Web service agent to achieve its purpose." Based on this definition, we can consider an orchestration as a director of a software agent (program) behavior [14]. The agent exposes a Web service that is available to other agents, the result returned by the agent consists of a series of calls to basic services and transformations on the data retrieved from the basic services used. Figure 1 illustrates a simple agent based orchestrations [15].

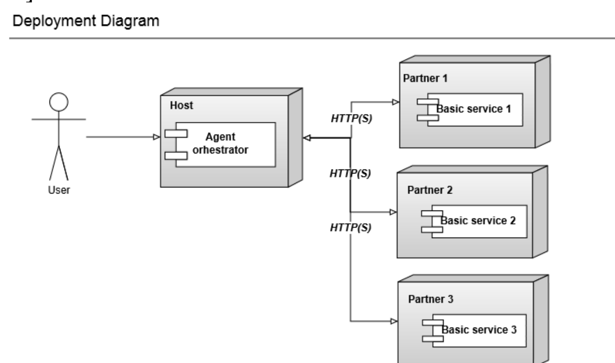

Figure 1. Connections of orchstration.

An orchestration gives rise to a semantic once interpreted. The benefit of orchestration is noticed during interpretation. The same semantics can come from many

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

styles of definitions. The model depends on the language used to implement the definition of an orchestration.

The approach that we present is an EIP based model of orchestration definition. The proposed model allows the building of orchestrations with semantics quite similar to those built by other models in the domain [16] [17] [18].

A. Business Agent

Our approach allows managing orchestrations composed of EIP's. In this section, we will see what a software agent; we will also see how to use an orchestration within an agent.

A business agent is an agent first. In addition, this agent assures the autonomy property. An agent is a program that is autonomous [19]. It has the ability to communicate with its environment and to perform the task for which it was made.

An agent is characterized by four main features:

- Autonomy: an agent is master of its decisions. Its behavior is not directed from the outside but it is self-managing agent. We can see the property of autonomy in two aspects: autonomy of the internal state of the agent and the autonomy of the agent's actions. Internal autonomy means that the agent is able to change its state by objective. The autonomy of action means that the agent is able to make a decision based on the information from its environment. Both aspects of the autonomy of the agent are provided by the π-DSL language. The ultimate goal of the agent is to compose a response to an invocation. This composition is based on communications with business and monitoring components.
- Reactivity: the agent is able to perceive the changes in its environment using the components of monitoring and possibly take action in response to changes in this environment.
- Proactivity: an agent is able to determine the actions to achieve its objective, it is based on its internal state and the information received from its runtime environment.
- Social: an agent is able to communicate with other agents, to carry out its mission and achieve its objective. Given that agents expose their services using the same interface type as the components business. Calls to agents and business services base happens in a transparent manner.

A business agent is a composition of business services characterized by four properties of the agent. These four properties are provided by our approach to defining business agent A. π-DSL.

B. EIP orchestration

Several EIP based specifications exist, which were not initially dedicated to Web services orchestration, but could be used as tools allowing orchestration, like Translator or Aggregator. We have decided to base our approach on these specifications. These EIP specifications are the base of the different interactions with basic services as well as the transformations necessary to build an orchestration. Thereby, orchestrating inherits the properties of the EIP that compose it. Note that the order of definition is important and must be preserved during execution.

EIPs provide a framework for interacting with partners to transform the data flow and be invoked by other partners. Each EIP provides a work step, i.e., interaction in the orchestration; it is possible to have a work step composed of several EIPs.

Given that the EIPs are based on the "pipe and filter" architecture, they automatically provide the concepts of channel messages, routing, transformation and endpoint. Messages are what travel between a pipe and a filter. The structure of a message is as specified in the JMS [20]. In this paper, a channel allows a message to transit and an endpoint is a destination of the message. In addition, EIPs introduce the concepts of routing and transformations between channels and endpoints.

Our orchestration will be a composition in which each step is based on one or more of EIP concepts.

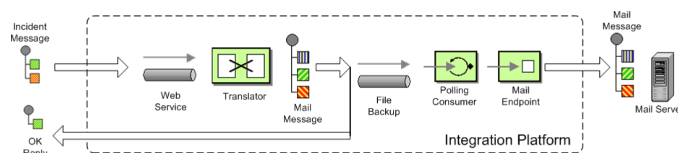

Figure 2. An EIP based system

Figure 2 shows some EIPs, and how it is possible to build an EIP from basic treatments. These treatments are basic bricks we use to define our orchestrations.

Our orchestrations are exposed as Web Services endpoints. When an exposed endpoint is invoked, the orchestration activates the different EIP component of the requested orchestration. Activation of an orchestration can allow data transformation, invoking the participants in this orchestration and returning a result to the client on the initiative of the invocation on the exposed endpoint.

Our system supports various treatments and activities offered by other systems, such as BPM orchestration. The difference lies in the fact that the treatments and activities are implemented within well-defined patterns.

III. FORMAL SPECIFICATIONS

In this section, we will present the formal specifications of our system. We will start by a reminder of the π-calculus language, then we will present and comment on some parts of the specifications of our system and finally, we will present an example of agent-based orchestration definition as a foundation of our case study.

A. π-calculus

The π-calculus is a formal language designed to define concurrent systems. The language basically focuses on the communication between parallel systems. The language was developed by R. Milner [21] and was published for the first time in [22]. The π-calculus is based on the concept of terms and names. Term represents a process or sub-process. Also, a term consists of a sequence of emissions and receptions via communication channels. It also consists of calls to other

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

terms. However, a name can be either a communication channel or a variable that will be calculated by the values received via a channel.

$$
\begin{aligned}
S &\overset{\text{def}}{=} (\nu\, c\, d)\,(P(c,d) | Q(c,d)) \\
P(c,d) &\overset{\text{def}}{=} (\nu\, b)\,\big(c(a).\tau.\bar{d}\langle b\rangle \big| \emptyset\big) \\
Q(c,d) &\overset{\text{def}}{=} (\nu\, a)\,(\bar{c}\langle a\rangle.\tau.d(b) | \emptyset)
\end{aligned} \quad (1)
$$

The equation (1) is a definition of *S*, a term that execute in parallel the term *P* and *Q* that use the canals *c* and *d* to communicate with each other. This definition is expressed using one of the three variations of the π-calculus, which is the monadic π-calculus. This variation characteristic is that a communication channel can transfer only a single value.

The second variation of the π-calculus is polyadic π-calculus. The main difference between the monadic and polyadic is that the latter can transmit and receive multiple names on the same channel as demonstrated in the (2) using the same example from term "S".

$$
\begin{aligned}
S &\overset{\text{def}}{=} (\nu\, c)\,\big(P(c) \big| Q(c)\big) \\
P(c) &\overset{\text{def}}{=} (\nu\, b)\,\big(c(a, callback).\tau.\overline{callback}\langle b\rangle \big| \emptyset\big) \\
Q(c) &\overset{\text{def}}{=} (\nu\, a\, callback) \\
&\quad (\bar{c}\langle a, callback\rangle.\tau.callback(b) | \emptyset)
\end{aligned} \quad (2)
$$

The third variation is the π-calculus of higher order. This variation contains all the characteristics of the polyadic π-calculus. In addition, it allows to send and receive terms and names via a channel in the same way. The equation (3) shows the transfer of a term 'R' between terms 'P' and 'Q'. Therefore, showing that the execution of the term 'R' is on the target process.

$$
\begin{aligned}
S &\overset{\text{def}}{=} (\nu\, c)\,(P(c) | Q(c)) \\
R(callback, param) &\overset{\text{def}}{=} \\
&\quad (\nu\, b\,)\,\big(param(v).\tau.\overline{callback}\langle v\rangle \big| \emptyset\big) \\
P(c) &\overset{\text{def}}{=} (\nu\, b) \\
&\quad \big(c(a, param, Process).\tau.\overline{param}\langle b\rangle | Process\big) \\
Q(c) &\overset{\text{def}}{=} (\nu\, a\, callback\, param) \\
&\quad (\bar{c}\langle a, param, R(\,callback, param)\rangle.\tau.callback(b) | \emptyset)
\end{aligned} \quad (3)
$$

We will use the extension communication operator [23] in a polyadic context as shown below:

$$x.F \mid \bar{x}.C \;\; \rightarrow F \bullet C \quad (4)$$

Let us define the following:

$$
\begin{aligned}
F &\overset{\text{def}}{=} (\lambda\, \vec{y})P \\
C &\overset{\text{def}}{=} [\vec{y}]Q
\end{aligned} \quad (5)
$$

The operator $-\bullet-$ allows us to define an interface between the two terms in which it operates. This will make possible to dynamically integrate terms with the entire orchestration steps. This operator can be assimilated to a communication interface in UML as shown in Figure 3.

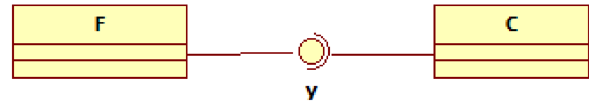

Figure 3. π-calculus interface

B. Construction of a definition of orchestration

We consider 'Orch' an orchestration with a single participant. The variable IN from (6) represents an input of the orchestration:

$$IN \overset{\text{def}}{=} (\nu\, y)[\vec{y}] \mid (\lambda\, \vec{z}).\tau \quad (6)$$

And the term OUT in (7) is the sole participant in the orchestration:

$$OUT \overset{\text{def}}{=} (\lambda\, \vec{z}).\tau \mid [\vec{y}] \quad (7)$$

The vector $\overrightarrow{Pr}$ represents all the terms corresponding to processing steps and transformations performed between receiving a request and returning the result. We can then define the term 'Orch' as follows:

$$\text{Orch} \overset{\text{def}}{=} IN \bullet \Big(\big((\lambda\, \vec{y})\; Pr_i\; [\vec{z}]\,\big) \bullet\Big)^{\|Pr\|} OUT \quad (8)$$

The term „Orch" given in (8) creates a flow through all terms Pr_i between the input 'IN' and the output 'OUT'. Each term Pr_i representing a step in the orchestration will have a vector of names as input. Each term will have a second vector Pr_i as output. These vectors will be transported between the different steps following the same order defined within the vector $\overrightarrow{Pr}$. The input $\vec{y}$ to the Term Pr_i is connected to the output $\vec{z}$ of the term Pr_{i-1} while its output is connected to $\vec{z}$ the input $\lambda\, \vec{y}$ of the term $Pr_{i\pm1}$.

The operator "•" is an ideal way to represent an exchange that carries the communication streams between two steps of an orchestration. This operator will help us to connect the various processes that define an orchestration.

As we have seen, our orchestrations are in the form of a set of steps (transformations) between an endpoint and the participants of the orchestration. The list of steps has not been known by the engine before loading the definition of orchestration. We will use a data structure in order to persist the definition of orchestration. The instance of this structure will be loaded by the engine via an activator that is a particular endpoint type for connecting managed services to an input channel. The engine will be based on this definition that it receives in the form of a linked structure to activate the orchestration.

Activation of the orchestration can link the different steps. As illustrated in Figure 4, the link between these steps is the connection of inlet flow of step 'n' with the exit of 'n-1' using the concept of exchange, which carries a two-way flow.

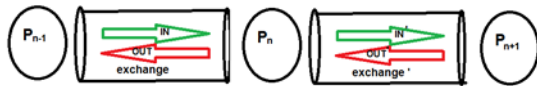

Figure 4. An EIP based system

We will use both π-calculus concepts of abstraction and concretion in order to implement dynamic linking on chained lists. These lists will be used to contain the different steps of our orchestration.

IV. An approach based on MDA

We defined the π-calculus language as meta-meta-model. In Section 5, we will present the definition of a meta-model in π-calculus. Meta-model consists of an extension of π-calculus as dedicated to DSL service orchestration based routes. Routes are an implementation of pipe and filter architecture using routing rules. The proposed DSL takes a form of a composition of EIP. Meta-model also describes the tools needed to run a model once created. These tools are in the form of a set of components. The models are created using the π-calculus based DSL. Figure 5 illustrates the four levels of our approach.

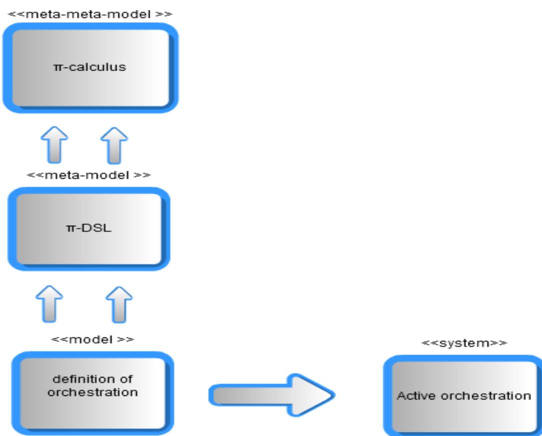

Figure 5. MDA Model

In the next section, we will detail the transformations made between the different models.

A. Model-driven orchestrations definition

Our approach in defining orchestrations is a MDA based approach [24]. The business area of our system is the definition of orchestrations; these orchestrations are components of the fundamental services. We have extracted domain-specific vocabulary as a π-DSL language. We can represent the π-DSL as a set of terms called EIP when EIP = {from, process, to ...}

Each orchestration will be defined using a language described in π-calculus. This language allows the interaction between various tools made available to the orchestrations.

Our meta-meta-model describes a language of orchestration in addition to the tool permitting the interpretation of this DSL orchestration language. The interpretation tools using π-DSL will be subject to a manual transformation [25] to object-oriented programming language [26]. The execution of the system supports different terms materialized from meta-meta-model in order to connect via the EIP channels. These channels are essential to the π-DSL.

Each orchestration is defined as a set of "emissions" on the EIP channels. Emissions existing on the EIP channels are received by one of the tools, which are the same as the term Routes that will be described in detail in subsequent section. We will also specify the term Route that allows transforming the definition of a π-DSL orchestration into a definition taking the form of data structure. This data structure represents the Platform independent model (PIM) [27] orchestration.

The structure representing the PIM is transformed in order to activate the orchestration. The step involving the activation transforms the structure representing the PIM in an executable code representing an orchestration language. The code will be generated automatically as Camel java-DSL [28]. The Camel DSL code communicates on the same channels as the EIP tools defined in the meta-meta-model.

Figure 6 illustrates an example of an orchestration that uses a service that transforms the Route of this service before returning it to the customer at the initiative of the invocation. Consumer and Provider are specific process wrappers for external endpoints interaction.

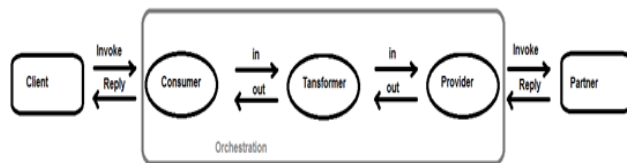

Figure 6. Exchages in orchestration

Our goal is to reach an executable system from the definition in the form of π-DSL. To do this, we perform a set of transformations whose outlines are highlighted in the Figure 7.

In the next section, we detail the structure of meta-meta-model orchestrations then in the next section, we will talk about the definition of the various EIP, which constitute the π-DSL routing and orchestration oriented language. Then, in the section dedicated to message route, we will detail the activation principle such as we designing our approach.

B. Model-driven orchestrations transformations

In our approach, the definition of orchestrations is the body of the wrapper agent of these orchestrations. Each agent has a definition, which characterizes it by an orchestration that is unique for the agent itself. Applying the definition of the agent in our system triggers a change in the system state. This new state is reached after the activation of

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

the orchestration definition. The activation implements the semantics described by the definition of orchestration.

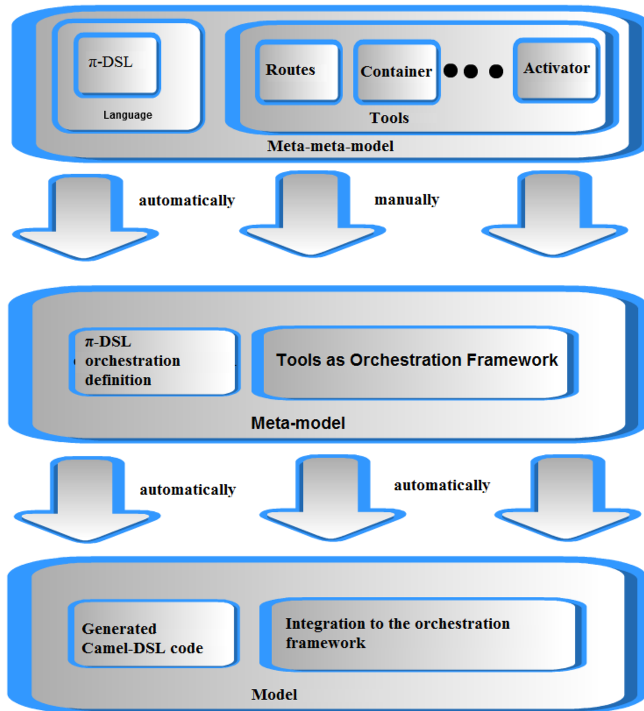

Figure 7. MDA transformations levels

Orchestrations will use the concept of route introduced by EIP. The Route is the building blocks of an orchestration. The Route is used to associate an input to transformations and outputs. Inputs are endpoints exposed by the agent while the outputs are endpoints consumed by the agent. Transformations can be applied to both input and output stream flows.

The Figure 8 shows an orchestration using the content based router EIP and message translator EIP to route the input message to the adequate translator

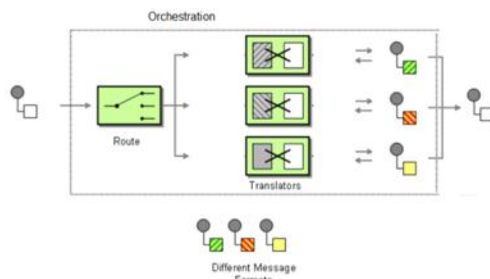

Figure 8. An EIP orchestration

The definition of an orchestration and the semantics of an orchestration are separate concepts. So far we have only discussed the definition of orchestration, which is composed of the series of actions to take in response to an external invocation. Each orchestration is a model. It is described using the π-DSL, which is the extension of π-calculus offered by the meta-meta-model (see Section 5).

The π-DSL consists of all the EIP channel names. It defines an orchestration through signals on EIP names. Since π-DSL is an extension of π-calculus, it inherits all its properties. This gives the possibility to manipulate some terms that are free within the π-calculus limitations. Manipulated terms will be called processors and will have at their disposal data streams they can use.

During the orchestration activation, the definition is transformed into an instance. Activation is made via a component that is one of the different tools defined in the meta-meta-model. These tools are defined as terms in the section dedicated to the definition of the system.

The definition of an orchestration considers the definition of a general context of the process as shown in Figure 9.This context allows the exchange of shared information between the various components of the orchestration. This set of shared variables is a part of the state context of the business agent at a given time. The result of the invocation of a route will depend on the current state of the agent because a previous invocation may have set a value on a shared variable, and thus influence the final result.

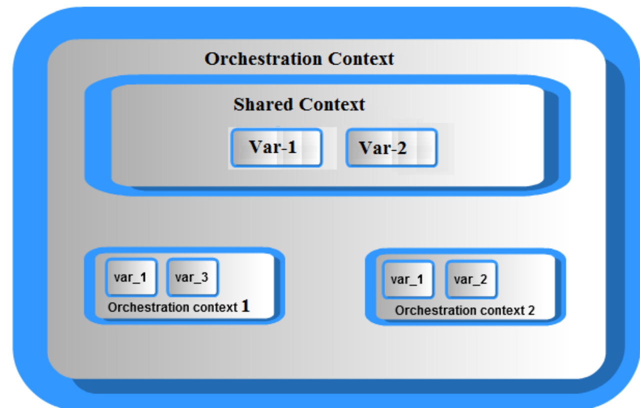

Figure 9. Shared context

The semantics of the agent is enhanced after loading the definition by the engine. The engine activates the orchestration routes and thus integrates the wrappers (Consumers and Providers). Then, the engine loads the context of the agent. Following this action, we end up with an active and ready-to-receive external invocations system state. However, it is important to make the distinction between the contexts of the agent corresponding to the internal information of the agent on one side and the state of the system that contains the context and the routes constituting the different agents on the system.

V. SYSTEM DEFINITION

Based on the definition (1), our system (9) defines a container running in parallel with the Repository.

$$System \stackrel{\text{def}}{=} (\nu\; repoGet\; repoPut)$$

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

$$Container(repoGet, repoPut) | \quad Repository(repoGet, repoPut) \tag{9}$$

The Repository (10) is a term that represents a composition for sharing the definition of agents. It can add an artifact containing the definition of an orchestration or retrieve the artifact using the URL that was used to add the artifact. The processing performed inside the Repository complies with the Maven [29] specifications. We will ignore the details of the inner workings in this paper.

$$\begin{aligned} Repository(repoGet, repoPut) \overset{\text{def}}{=} & \ \overline{repoGet}(paxuri_i, http).\tau \\ & .\overline{http}\langle bundle_i\rangle.Repository \\ & + repoPut(paxuri_j, http) \\ & .http(bundle_j).\tau.Repository \end{aligned} \tag{10}$$

The container (11) is the container application on which our services and our agents will be deployed. It allows loading definitions of orchestrations in its context. The container and the system have the same execution context.

A container can host any number of agents and services. Because each agent/service has a definition of its own, let's take the example of a system that contains one agent that performs an orchestration using a couple of services. The container allows the sharing of different channels to activate the definition of an agent in the engine.

Shared channels are associated with EIP. The definition of orchestration is transformed after activation in a set of Routes respecting an EIP sequence.

In order not to overload our definitions with a large number of parameters we will use the name "EIP" to represent all EIP names.

$$\begin{aligned} & Container(repoGet, repoPut) \overset{\text{def}}{=} \\ & (\nu\ EIP)\ (\ (\nu\ l, install, start, uninstall\ stop) \\ & (Runtime(l, install, uninstall\ ) \\ & |\ Engine(l, EIP, start, stop)) \\ & |(\nu\ uri_a,\ uri_b) \\ & \ (Agent(EIP)|ServiceA(uri_a)|ServiceB(uri_b))\ ) \end{aligned} \tag{11}$$

Runtime (12) is designed to: manage the retrieval, activation and shutdown of various artifacts containing the definition of the agent as well as services. For this, it communicates with the Repository to recover the definition using the URL of the artifact. Once the artifact is recovered, it executes the definition to activate the engine.

$$\begin{aligned} Runtime(l, install, uninstall) \overset{\text{def}}{=} & \ (\nu\ \ http, r) \\ & (install(paxUrl, status) \\ & .repoPut(paxUrl, http).http(bundle_i) \\ & .MapPut(l, bundle_i, r).r(id).\overline{status}\langle id\rangle) \\ & + (\nu\ \ http)(\ uninstall(id, status) \\ & .MapRem(id, r).\overline{status}\langle id\rangle) \end{aligned} \tag{12}$$

The Engine (13) enables The Routes activation. Routes will be added to the system’s context. The integration of context changes their status. The new status supports invocation of the active orchestration.

$$\begin{aligned} & Engine(EIP) \overset{\text{def}}{=} \ (\nu\ l) \\ & \quad Routes(l, EIP)|\ RouteActivator(l) \\ & RouteActivator(l) \overset{\text{def}}{=} \ (Processeur(l_i)\ \bullet)^{\|l\|} \\ & \ |\ (\nu\ \ http)(\ start(id, status).MapPut(id, l) \\ & \ .r(bundle_j).\overline{status}\langle id\rangle.RouteActivator(l) \\ & \ + (\nu\ \ http)(\ stop(id, status).MapRem(id, r) \\ & \ .r(bundle_j).\overline{status}\langle id\rangle.RouteActivator(l) \end{aligned} \tag{13}$$

The term Routes (14) is the basic element of the activation of an orchestration, as the term that uses the "emissions" on EIP channels. It is able to add to the system the ability to run the orchestration, then, transform this definition to a set of steps that are executed after the event fired.

$$Routes(l, EIP) \overset{\text{def}}{=} from(uri).Route(l, uri, EIP) \tag{14}$$

The term Route (15), as its name suggests, allows you to link an entry to one or more outputs. Routing the term can manage a set of connections between both ends with a transform in the stream exchanged if needed.

$$\begin{aligned} & Route(l, uri, EIP) \overset{\text{def}}{=} \\ & \quad process(P).MapPut(l, Processeur\ (P)) \\ & \quad + from(uri).MapPut(l, Consumer(uri)) \\ & \quad + to(uri).MapPut(l, Provider(uri)) \end{aligned} \tag{15}$$

The first step is the transformation of a π-DSL definition to data structure representing an orchestration. This transformation is conducted by the term 'Routes' listening on the EIP channels. At each "emissions", the term Route manages the integration of a Route in the current orchestration. To do this, the term 'Routes' Delegates the treatment of integration PIEs to orchestration. Therefore, appealed to the term Route after each transmission on channel EIP 'from'.

The second level of transformation is the transformation of the structure representing a Route in a set of processes chained together and able to implement the semantics of the orchestration

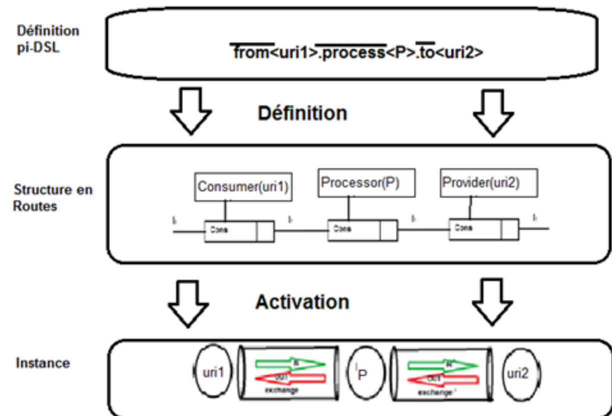

Figure 10. Activation of orchestration

This subdivision illustrated in Figure 10 allows us to keep control of an intermediate data structure, which may be modified to adapt it to the target platform. This transformation is at the heart of the migration mechanism that we will detail in a future paper

VI. CASE STUDIES

In order to illustrate our approach by case studies, we will take as an example the definition of an orchestration between two weather services and compare the values returned by called services.

We begin by defining our orchestration that will be as shown in Figure 11:

$$\overline{from}\langle uri_1\rangle.\overline{process}\langle P_1\rangle.\overline{to}\langle uri_2\rangle.\overline{process}\langle P_2\rangle.\overline{to}\langle uri_3\rangle$$

Figure 11. Generated Camel-DSL code

This definition is subject to an automatic transformation (as shown in Figure 7) of π-DSL part, against the terms {P1, P2} that represent the processor, which will be subject to manual transformation.

A mapping is defined between the pair {P1, P2} and there collocations in a π-DSL definition. The result will be in the form of Camel DSL code ready to be loaded and run on tools materialized from the meta-meta-model. Tools are generated in the form of a container, which uses Apache Felix [30] as a basis for implementing the definition of the container.

The second tool is the repository, which is an implementation standard Apache maven.

The third is the runtime that is included in the OSGi container (Felix) and provides a shell "Gogo" for interacting with the external.

Go back to our example of the definition of agent orchestration. The transformation from the π-DSL in code "Camel-DSL" leads to a deployable artifact on the container. The code is as shown in Figure 12.

```
import org.apache.camel.builder.RouteBuilder;
/**
 * A Camel Java DSL Orchestration
 */
public class OrchestrationRouteBuilder extends
RouteBuilder {

    public void configure() {

        from("nmr:uri1")
               .process(p1)
                   .to("nmr:uri2")
               .process(p2)
                   .to("nmr:uri3");
    }

}
```

Figure 12. Generated Camel-DSL code

Once deployed and activated, this route allows us to integrate the services present on the uri2 and 3 with the client that invoked the uri1.

The Camel engine will take control of the artifact deployed and ensure the interpretation of the Camel-DSL code. The engine will incorporate routes contained in the artifact to its execution context. The result will change the state of the system initially defined by the tools generate during the transition from meta-meta-meta-model to model.

The system is then enriched by the definition of the agent. Activation of this definition enhances the overall execution context.

VII. CONCLUSION

In this paper, we were able to develop an approach for generating a system dedicated to the orchestrations. Our approach is based on the MDA approach to obtain a dedicated orchestration and a set of tools constituting the execution context of the π-DSL orchestration

The formalism represented by the π-DSL language, defines an orchestration as a composition EIP. The orchestration is transformed into a camel-DSL and packaged as Maven artifact. The activation of the archetype load routes EIP composes orchestration.

We will discuss in a forthcoming paper on mobility in order to include in the definition of our system. We will prove by model checking [31] the mobility support of the system code.

We propose an extension of the semantics of our approach by adding a new dimension of freedom through the mobility aspect, which will be added to the semantics of an orchestration.

REFERENCES

[1] C. Peltz, “Web Services Orestrestration and Choreography,” Computer, vol. 36, no. 10, Oct. 2003, pp. 46-52

[2] BPMN. Bpmn - business process modeling notation. ‘http://www.bpmn.org/ retrieved: October, 2013

[3] C. Ibsen and J. Anstey, Camel in Action, Manning Publications, 2010

[4] C. Walls, R. Breidenbach, Spring in Action, 2nd Ed, Manning Publications, 2008

[5] M. Endrei et al., Patterns: service-oriented architecture and web services. IBM Corporation, International Technical Support Organization. 2004.

[6] G. Hohpe and B. Woolf, Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions . Addison-Wesley, Boston, 2004.

[7] D. Chappell, Enterprise Service Bus, O‟Reilly Media, Inc., Sebastopol, 2004.

[8] A.Charfi and M. Mezini, “Hybrid Web service composition: business processes meet business rules,” Proc. ICSOC ‟04, Proceed- ings of the 2nd international conference on Service oriented computing, ACM Press, New York, 2004, pp. 30–38.

[9] D. B. Lange and M. Oshima, “Seven good reasons for mobile agents,” Commun. ACM , vol. 42(3), 1999, pp. 88–89.

[10] C. Mahmoudi and F. Mourlin, “Adaptivity of Business Process,” Proc. ICONS 2013, The Eighth International

[11] OSGi Alliance. OSGi Service Platform Core Specification , release 4, version 4.2 ed. 2009 http://www. osgi.org retrieved: October, 2013.

Copyright (c) IARIA, 2013.
ISBN: 978-1-61208-304-9

[12] G. B. Laleci et al., “A Platform for Agent Behavior Design and Multi Agent Orchestration,” Agent-Oriented Software Engineering Workshop, the Third Inter- national Joint Conference on Autonomous Agents & Multi- Agent Systems, 2004, pp 205–220.

[13] BonitaSoft. Bonitasoft : open source business process management and workflow software. URL : http://www.bonitasoft.com/ Retrieved on January 25, 2013

[14] T. Erl, Service-Oriented Architecture (SOA): Concepts, Technology, and Design ; Prentice-Hall, 2005.

[15] S. P. Fonseca, M. L. Griss, and R. Letsinger, “Agent behavior architectures a MAS framework comparison,” Proc. AAMAS, 2002, pp. 86–87.

[16] M. Viroli, E. Denti, and A. Ricci, “Engineering a BPEL orchestration engine as a multi-agent system,” Journal of Science of Computer Programming, vol 66, issue 3, 2007, pp. 226-245.

[17] A. Charfi and M. Mezini, “Aspect-oriented web service composition with AO4BPEL,” ECOWS, LNCS, vol.

[18] D. Jordan and J. Evdemon editors. Web services business process execution language version 2.0.http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf retrieved: October, 2013

[19] F. Stan and A. Graesser, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents." Intelligent agents III agent theories, architectures, and languages. Springer Berlin Heidelberg, 1997, pp. 21-35.

[20] R. Monson-Haefel and D. Chappell, Java Message Services. O’Reilly, 2001.

[21] R. Milner, The polyadic p-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Also in Logic and Algebra of Specification, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

[22] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II . Volume 100 of Journal of Information and Computation , pages 1-40 and 41-77, 1992.

[23] D. Sangiorgi, “From -calculus to Higher-Order -calculus | and back,” Proc. TAPSOFT, LNCS 668 . Springer-Verlag, 1993.

[24] A. Kleppe, S. Warmer, and W. Bast, MDA Explained. The Model Driven Architecture: Practice and Promise, Addison- Wesley, April 2003.

[25] S. R. Judson, R. B. France, and D. L. Carver, Specifying Model Transformation at the Metamodel Level, Wisme 2003.

[26] M. Campione and K. Walrath, The Java Tutorial. Addision-Wesley, 2003.

[27] G. Benguria, X. Larrucea, B. Elvesæter, T. Neple, A. Beardsmore, and M. Friess, “A platform-independent model for service-oriented architectures,” Proc. I-ESA‟06, 2006.

[28] R. Z. Frantz, “A DSL for enterprise application integration,” International Journal of Computer Applications in Technology, vol. 33(4), 2008, pp. 257–263.

[29] Maven , In Apache Maven Project, http://maven.apache.org/ Retrieved on January 25, 2013

[30] Apache felix. http://felix.apache.org/site/index.html Retrieved on January 25, 2013.

[31] B. Bérard et al., “Systems and Software Verification,” Model-Chec king Techniques and Tools, Springer, 2001.