A System Organic Architecture based on Dynamic Functional Architecture Modeling

Jacques Simonin, Pierre-Yves Pillain

A System Organic Architecture based on Dynamic Functional Architecture Modeling

Jacques Simonin and Pierre-Yves Pillain
Institute Mines-Telecom Atlantique
Lab-STICC UMR CNRS 6285, UBL
29238 Brest, France
jacques.simonin@imt-atlantique.fr

***Abstract*—We focus on dynamic model usability and utility, performed throughout system engineering. The dynamic model is designed in relation to the system use scenarios. We generate then automatically the static models (data model, component model) from the dynamic one. A dynamic functional modeling based method is proposed to generate automatically the static and the dynamic organic architecture of a system. The method consists of the definition of system engineering rules constraining the transformation of a dynamic functional model of the system into static and dynamic organic models. The system engineering rules conform to a 3-layers organic architecture. The implementation of this transformation with operational-QVT allows an automatic generation of the organic models. The illustration of the transformation concerns a system of ordering. This work discusses on the relevancy of this method based on Model Driven Engineering.**

Keywords—system engineering; model based engineering; dynamic modeling; static modeling; functional architecture; organic architecture.

I. INTRODUCTION

Improving system design processes can be done either with tools or models that are meant to support this process, but also through the adaptation of system and software engineering processes (as specified for example in ISO/IEC/IEEE 15288 for system [1] or in ISO/IEC 12207 for software [2]). User involvement is one of the known keys of success. In most of system engineering methodologies, the functional analysis of the user needs is first specified. Then, during the design, the developers have to define each system part or how each function will perform.

System Engineering (SE) [3] suggests a structured process to build complex systems. SE consists in specifying needs or requirement analysis ("what"), proposing a solution or architectural design and implementation ("how"), integration, verification, validation of this solution and then its maintenance. Several processes described in the ISO standard associated to system or software life cycle, or UP (Unified Process) [4] are necessary to perform the project. SE uses various formalisms to model functional and organic architectures, including static and dynamic points of view. The dynamic point of view is defined by the situation scenario of the static one for all the interactions with the system (triggered by another system or a user).

Some basic activities are commonly recognized: stakeholder needs elicitation of requirements, functional analysis, architectures and coding. Activity artefacts as described in [3] are:

1) During stakeholder needs collection, the functional and non- functional requirements have to be identified for the system that must be developed.
2) Functional analysis results from the system specification. The use cases describe what the system has to perform. The scenarios which illustrate the use cases define a dynamic use of them.
3) Functional architecture is designed from functional requirement specification. As such, functional components participate in the dynamic use of the system.
4) Technical architecture satisfies the collected non-functional requirements and is described by nodes and execution environment.
5) Organic architecture, in relation to SE, is based on the design of the components' interfaces. These interfaces implement the functional architecture of the system and components are deployed on the technical architecture.

These activities are defined in the EA4UP (Enterprise Architecture for Unified Process) [5] method, which constrains a system engineering with EA (Enterprise Architecture) recommendations [6]. This mapping of the architecture design of a system with the architecture design of its IS (Information System) is based on MDE (Model Driven Engineering).

Our contribution is twofold. It proposes (i) a dynamic meta-modeling of the functional architecture and a dynamic meta-modeling of the organic architecture of a system and (ii) a method allowing the transformation of dynamic functional modeling into an organic one. The transformation conforms to a chosen 3-layers organic architecture. The organic static models (data model and component model) are automatically generated from the dynamic one.

Section III presents the transformation (meta-modeling and architecture rules), which results in an organic architecture model. In Section IV, the organic architecture models resulting

automatically from the previous implemented transformation are illustrated. The discussion in Section V focuses on the MDE use and its complexity in relation to a gap analysis between automated and manual organic architecture designs. Section VI concludes this contribution and presents some perspectives.

II. Related Work

The challenge when performing functional architecture description is the design of the internal behavior of the system that carries out the desired processes. The goal is the building of the internal system description as complete as possible (completeness). Designers and coders have to define "how" the system will work. Systemic approach [7] defines a system as interactions between unities. SE has its own language to support the functional description of such interactions. In the SE community, SysML® (Systems Modeling Language), EFFBD (Enhanced Functional Flow Block Diagram) [8] are current languages to aim the functional architecture description.

Our work focuses on software development in an IS. In this perspective, the definition of the processes is central in the needs specification and functional architecture description. The system supports indeed a business process. Business Process Management (BPM), including its notation (BPMN™: Business Process Model and Notation) [9], is a well-established practice. BPEL™ (Business Process Execution Language) [10] enables the business process execution thanks to Web Services (WS). Another way for describing the functional architecture is based on the service paradigm as defined in a Service-Oriented Architecture (SOA). SOA specifies that system entities (people, organization, systems) provide services to each other via asynchronous message transmission. SoaML® (Service oriented architecture Modeling Language) [11] enables to model this kind of architecture. SOA model, which describes a functional architecture, can result from:

1) A BPMN model [12].
2) A set of use cases specifying the system [13].

The Service-Oriented Computing (SOC) approach promotes the idea of assembling components into a network of services that can be loosely coupled to create flexible, dynamic business processes and agile applications that span organisations and platforms" [14]. SOA is the way to describe this vision. In this approach, the goal is to separate the definition of the service from their implementation.

To facilitate this work, we propose to focus the transformation resulting in the organic architecture of a system and to specify the meta-models and the rules which define this transformation.

III. Dynamic Functional Architecture into Organic Architecture Transformation

The automation of the organic architecture design results mainly from a transformation of models. The model transformation needs a meta-modeling defining the useful concepts for the dynamic functional architecture and for the dynamic organic architecture of a system. It is completed by a meta-modeling of the technical architecture of the system and by the organic Enterprise Architecture of the IS. Meta-models concerning functional and organic architecture are designed in relation with the same concept: the scenario which enables to describe a dynamic behavior.

A. Dynamic Functional Architecture Meta-Model

In the functional architecture meta-model, the interactions defined by a sequence of functional messages useful to a scenario are the basis of the dynamic functional architecture. Scenarios can be executed in parallel.

A Functional Architecture Model (FAM) satisfies the meta-model represented by the Ecore [15] diagram in Fig. 1.

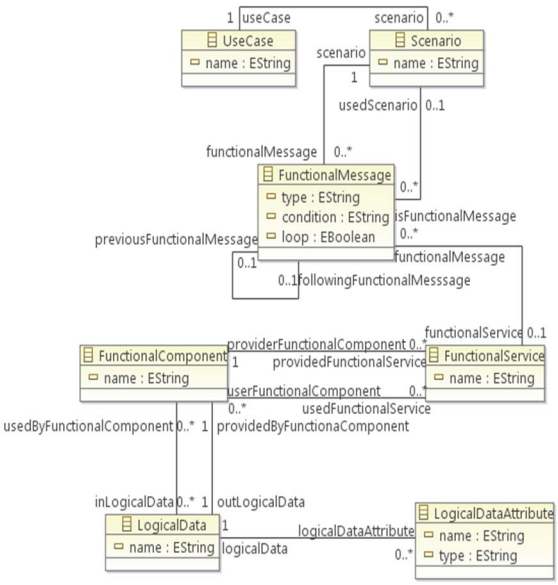

Fig. 1. Ecore diagram representing the dynamic functional architecture meta-model.

A sequence of functional messages (*FunctionalMessage* EClass) makes up the dynamic behavior of the functional architecture of a system. Each sequence supports a scenario (*Scenario* EClass) specified during the activity of functional analysis. A scenario illustrates a use case (*UseCase* EClass).

A functional message can be addressed to a scenario (in case of reuse of an existing scenario) or execute a function (*FunctionalService* EClass). The message is (*type* EAttribute of *FunctionalMessage*) (i) a request, if the message is a call, or (ii) a resource if the message is an answer to the call. Each function is provided or used by a functional component (*FunctionalComponent* EClass). Moreover, a condition (*condition* EAttribute of *FunctionalMessage*) and a loop (*loop* EAttribute of *FunctionalMessage*) can be associated to a functional message. The condition means a test enabling the functional message and the loop means that the functional message is iterated. A function is based on elementary operations (Create, Read, Update, and Delete (CRUD)). A functional message typed request and executing a functional service is designed in relation to a scenario of the use case of the system. This functional message is addressed to the

functional component providing the service. The concept of data (*LogicalData* EClass) provided by the functional components and the concept of data attribute (*LogicalDataAttribute* EClass) complete the meta-model.

A model conforming to this functional dynamic meta-model is illustrated in Fig. 2 by a scenario illustrating the *UcOrderWithCustomerManagement* use case of a commercial system specifying the order of one or more products for a new customer. The scenario *ScOrderWithCustomerManagement* is the following one:

1) *Collect the customer information*.
2) *Collect the list of products wished by the customer.*
3) *Create the associated order*.

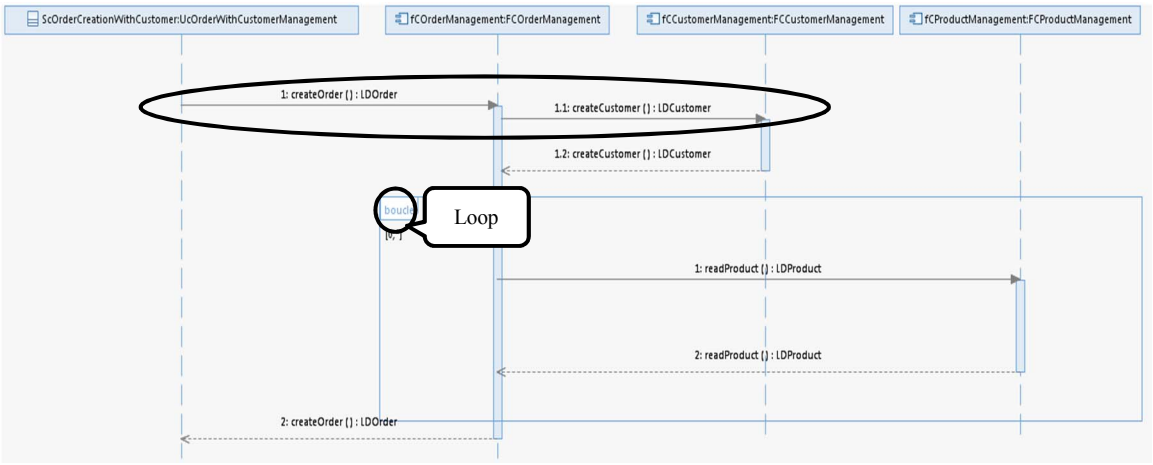

Fig. 2. UML sequence diagram representing the functional dynamic model carrying out the *ScOrderWithCustomerManagement* scenario including an example of a component dependency from FCOrderManagement to FCCustomerManagement consistent with the surrounded sequence of request typed messages.

The UML® (Unified Modeling Language) [16] sequence diagram in Fig. 2 representing the functional dynamic model supporting this scenario may be reformulated as:

1) The design of three logical data (defined by their attributes): *LDOrder* (*date*), *LDProduct* (*name*) and *LDCustomer* (*name*, *postalcode*).
2) The reuse of three functional components (defined by their operations) designed by functional enterprise architects: *FCOrderManagement* (*createOrder*), *FCProductManagement* (*readProduct*), *FCCustomerManagement* (*createCustomer*).
3) The design of six functional messages (three request typed and three resource typed) in the sequence diagram including a loop (UML combined fragment in Fig. 2) for the *readProduct* instance.
4) The design of the functional component dependencies resulting from the sequence diagrams and that conform to the functional enterprise architecture: *FCOrderManagement* on *FCProductManagement*, *FCOrderManagement* on *FCCustomerManagement*.

The illustration shows that the relationships (static) between functional components are consistent with their interactions (dynamic). For example, the functional message sequence composed by the two first request typed functional messages of the sequence diagram (see surrounded sequence of functional messages in Fig. 2) is consistent with the functional component dependency from *FCOrderManagement* (containing *createOrder*) on *FCCustomerManagement* (containing *createCustomer*).

A transformation resulting in an organic model needs also a technical architecture model in input.

B. Technical Architecture Meta-Model

A Technical Architecture Model (TAM) is actually very simple because the experiments do not need more complexity. The meta-model specifies the type of the technical infrastructure. This type is useful to allow a reuse of an organic element. An organic element can be reused only if its deployment on a technical infrastructure is consistent with the type of the technical infrastructure of the developed system.

The illustration of the technical architecture model is a JEE environment, which defines the developed system.

The last transformation input, which fosters the reuse, is a model of recommendations of the organic EA of the IS, which contains the developed system.

C. Organic Enterprise Architecture Meta-Model

An Organic Enterprise Architecture Model (OEAM) is usually a list of organic services, which are made available for the development of any systems of this IS. OEAM conforms to the meta-model composed by the following concepts. An organic service (*EAOrganicService*) is reusable if the implemented function (*EAFunctionalService*), based on elementary operations (Create, Read, Update, Delete), and the technical infrastructure where it is deployed on (*EATechnicalInfrastructure*) conform to the functional architecture and the technical architecture of the developed system. An EA service reuse, which is applied to the dynamic organic architecture of the developed system, needs indeed common carried out functional services and common deployment technical infrastructure. Moreover, a reuse of an organic service needs the knowledge of the component (*EAOrganicServiceComponent*) that provides it.

The illustration of the organic EA characterizes a commercial IS. A service allowing the reading a product is available in this IS. The *OSReadProduct* EA organic service implements the *readProduct* functional service. It belongs to the *OSCProductManagement* organic service component, which belongs to the commercial IS. This component is deployed on a *JEE* infrastructure and could be reused during the dynamic organic architecture.

D. Dynamic Organic Architecture Meta-Model

The output model of the transformation is an organic architecture model (OAM). The associated meta-model is represented in Fig. 3.

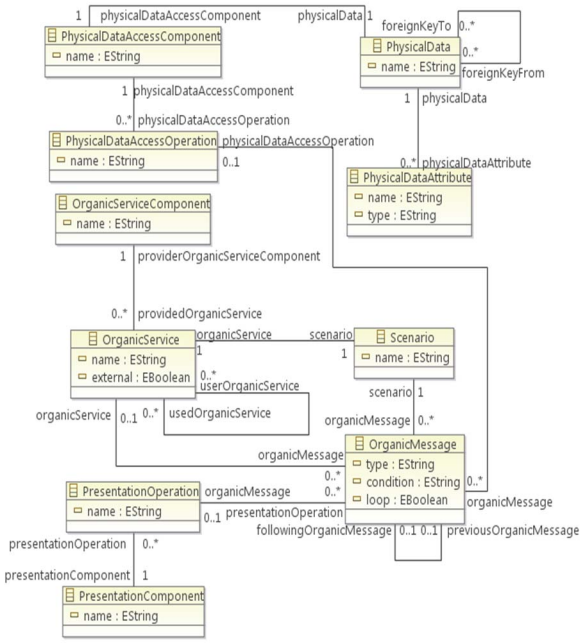

Fig. 3. Ecore diagram representing the dynamic organic architecture meta-model.

The 3-layers organic architecture is based on: the "Presentation" layer responsible for the interactions between the user and the system, the "Service" layer responsible for the organic services provided to the user, the "Data access" layer responsible of the object-relational mapping due to a relational database. This case study enables to show the feasibility of an automated organic architecture design.

Organic messages (*OrganicMessage* EClass) sequence is the core of a dynamic OAM. As defined in functional architecture, each sequence supports a scenario (*Scenario* EClass) resulting from the functional analysis of a system. The characteristics (*type*, *condition,* and *loop)* of an organic message are identical to those of the functional message. These messages conform to the chosen architecture. The first kind of instantiation is those of an operation of the "Presentation" layer (*PresentationOperation* EClass), which is encapsulated in a component (*PresentationComponent* EClass). The second one targets the "Service" layer with an operation representing an organic service (*OrganicService* EClass) encapsulated in an organic component (*OrganicServiceComponent* EClass). The last one is the instantiation of an operation to access to a data (*PhysicalDataAccessOperation* EClass), which is encapsulated in a component (*PhysicalDataAccessComponent* EClass). An accessed datum is a physical datum (*PhysicalData* EClass), which is defined by some attributes (*PhysicalDataAttribute* EClass) and some foreign keys showing a relationship with other physical data.

This organic dynamic model results from a transformation which implements the architecture rules that are specified below.

E. Organic Architecture Transformation Rules

The following architecture rules support the transformation of a FAM into an OAM deployed on a TAM and mapped with an OEAM. The chosen 3-layers organic architecture is defined explicitly by these rules. Used concepts and their associations are defined in the previous meta-models.

The three following architecture rules enable the design of a component of a layer.

PresentationLayerComponentDesignRule. One presentation operation (*PO)* results from the transformation of one scenario (*SC)*. A presentation component, which provides *PO*, results from the transformation of a use case illustrating *SC* and triggered by a user (not included in another use case).

ServiceLayerComponentDesignRule. One organic service (*OS*) results from the transformation of one scenario (*SC*). An organic service component providing *OS* results from the transformation of the use case illustrating *SC*.

DataAccessLayerComponentDesignRule. One physical data access component (*DA*) results from the transformation of one logical data (*LD*). One physical data access operation encapsulated in *DA* results from the transformation of a functional service provided by the functional component producing *LD*.

The two following rules underline the chosen architecture pattern: (i) a component from the "Presentation" layer depends on a component from the "Service" layer, (ii) a component from the "Service" layer depends on a component from the "Data access" layer, but (iii) a component from the "Presentation" layer does not depend directly on a component from the "Data access" layer.

PresentationLayerOnServiceLayerDependencyDesign Rule. The dependency from a presentation component (resulting from a use case (*UC1*): see **PresentationLayerComponentDesignRule**) on an organic service component" (resulting from a use case (*UC2*): see **ServiceLayerComponentDesignRule**) results from the sharing of the same use case (*UC1=UC2*).

ServiceLayerOnDataAccessLayerDependencyDesign Rule. The dependency from an organic service component (resulting from a use case (*UC*): see **ServiceLayerComponentDesignRule**) on a physical data access component (resulting from a logical data (*LD*): see **DataAccessLayerComponentDesignRule)** results from the instantiation of the functional component (*FC*) providing *LD* during the running of a scenario illustrating *UC*.

This instantiation is specified by a functional message instantiating a functional service encapsulated in *FC*.

The two following rules allow the reuse of an organic service.

EnterpriseArchitectureOrganicServiceReuseRule. The reuse of an EA organic service (*EAOS*) during the running of a scenario (*SC*) results from the mapping of the EA functional service set implemented by *EAOS* with the functions services instantiated during *SC* and from the consistency of the technical deployment of *EAOS* with the technical deployment of the developed system.

OrganicServiceReuseRule. The reuse of an organic service (resulting from a used scenario (*USC*) of the developed system: see **ServiceLayerComponentDesignRule**) during the running of a scenario (*SC*) results from the inclusion of *USC* into *SC*.

The following rule is a design rule of a physical data.

PhysicalDataDesignRule. One physical data (*PD*) results from the transformation of one logical data (*LD*). A foreign key from *PD* to a physical data, which results from a logical data used by the functional component producing *LD* (see *usedByFunctionalComponent* EReference, of the association from the *LogicalData* EClass to the *FunctionalComponent* EClass, in Fig. 1), completes the transformation.

The three following rules specify the generation of an organic message.

OrganicMessageDesignRule. One organic message results from the transformation of one functional message, except, for those associated to the resulting presentation operation (see **PresentationLayerComponentDesignRule**) and the resulting organic service (see **ServiceLayerComponentDesignRule**) in relation to each scenario. The resulting organic message can be associated to a used organic service or a physical data access operation.

OrganicMessageLoopDesignRule. The loop characteristic of an organic message (resulting from a functional message (*FM*): see **OrganicMessageDesignRule**) results from the loop characteristic of *FM*. Moreover, a joint physical data, which results from the transformation of two successive functional messages (see **OrganicMessageDesignRule**), is designed if:

1) The two successive functional messages are request typed.
2) The second functional message owns a loop characteristic and not the first one.
3) The first functional message instantiates a functional service provided by a functional component that uses a functional service (see *usedFunctionalService* EReference, of the association from *FunctionalComponent* EClass to *FunctionalService* EClass, in Fig. 1) instantiated by the second functional message.

OrganicMessageConditionDesignRule. The condition associated to an organic message (*OM*) (resulting from a functional message (*FM*): see **OrganicMessageDesignRule**) results from a condition associated to *FM*. This condition is moreover transformed into one request and one resource typed organic messages. These messages target a data access operation, which enables to check if the condition is satisfied.

The last rule concerns the organic message sequence. The organic architecture is based on a temporal sequence of messages, while the functional dynamic architecture is based on dependencies instantiated during system use scenarios.

OrganicMessageSequenceDesignRule. The use of a functional service (*FS*) by a functional component (*FC*) (see *usedFunctionalService* EReference, of the association from *FunctionalComponent* EClass to *FunctionalService* EClass, in Fig. 1) is transformed into a sequence of two pair of organic messages (request and resource). The first organic message pair of the sequence results from a functional message instantiating *FS,* and, the second one results from a functional message, which instantiates a functional service encapsulated in *FC*. If the first functional message pair owns a loop characteristic, then the resulting organic message pairs (including the instantiation of the relevant joint physical data access operation) are reported at the end of the sequence.

F. Dynamic Organic Model Transformation Algorithm

The automatic generation of a sequence of organic messages associated to a scenario from the functional messages associated to the same scenario needs the following algorithm. This algorithm is adapted to the chosen 3-layers organic architecture.

For each system use scenario:

1) Design of groups of functional messages typed as request based on a continuous sequence of messages that belongs to a same group.
2) Design of a vector of functional messages typed as request satisfying the dependency previously defined, except in case of loop, which is reported at the end of the vector.
3) Design of the organic message typed as request instantiating the presentation operation.
4) Design of the organic message typed as request instantiating the service associated to the scenario.
5) For each functional message of the vector:
    a. Design of the organic messages typed as request, and as resource, for each used organic service (external or internal), if detected.
    b. Design of the organic messages typed as request, and as resource, for each physical data access operation, else.
6) Design of the organic message typed as resource instantiating the service associated to the scenario.
7) Design of the organic message typed as request instantiating the presentation operation.

IV. AUTOMATED ORGANIC ARCHITECTURE ILLUSTRATION

The *fsd2osd* core transformation implements the rules specified in III.E in relation to the meta-models designed in Section III. It is completed by transformations (see Fig. 4) appropriate to a modeling language or a coding language.

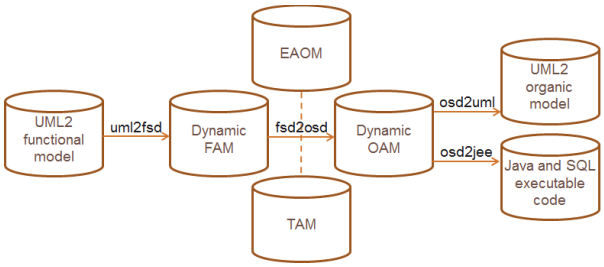

Fig. 4. Experiment transformations implementing the automated organic architecture design and code generation.

A. Dynamic Organic Model Transformation Tooling

UML 2.2 [16] (sequence diagram for dynamic modeling, class diagram and component diagram for static modeling) is associated to the transformation tooling. Three complementary model transformations are thus:

1) The *uml2fsd* transformation enables a conversion of a UML model, which conforms to the UML meta-model, including
   a. A set of classes representing the logical data and their attributes.
   b. A set of components, including their operations and the associated output parameter, representing the functional components, including their functional services and the produced logical data.
   c. A set of sequence diagrams representing the system use scenarios and instantiating the functional components.

   into a dynamic functional architecture model, which conforms to the FAM meta-model (see III.A).

2) The *osd2uml* transformation enables a conversion of an OAM model (see meta-model in III.D) into a model conforming to the UML meta-model. This transformation produces:
   a. A set of physical data, including their attributes and their foreign keys.
   b. A set of organic components associated to organic architecture layers and their operations
   c. A set of sequence diagrams representing the system use scenarios and instantiating the organic components consistently with the layer pattern.

3) The *osd2jee* transformation, which is not illustrated here, converts an OAM model into EJB code of organic services and database generation SQL script.

B. Dynamic Organic Architecture Model Illustration

The organic sequence diagram excerpts, which are presented below, results from a drag and drop of the selected UML interaction depicted in the project explorer to the pallet of the UML modeling tool. The algorithm (see subsection III.F) application to this OAM illustration targets the scenario *ScOrderWithCustomerManagement* (see subsection III.A):

1) Two groups of request typed functional messages are designed: {*createOrder*, *readProduct*} and {*createCustomer*}.
2) The vector of request typed functional message (with the instantiated functional service) is: {*createOrder createCustomer*, *readProduct*}, because *createCustomer* is used by *FCOrderManagement*, which encapsulates *createOrder*, and because the request typed functional message *readProduct* owns a loop characteristic.
3) Design of a request typed organic message instantiating a presentation operation (*orderCreationWithCustomerCall*) and of an organic message of request instantiating an organic service (*OSOrderCreationWithCustomer*) (see Fig. 5).

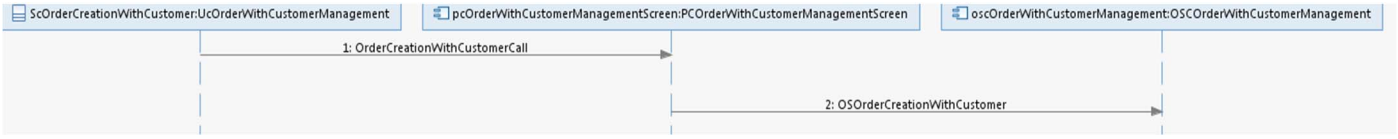

Fig. 5. Excerpts (see 3) of subsection IV.B) of the UML organic sequence diagram automatically generated.

4) Design of the organic messages implementing the vector of functional messages (see Fig. 6).
5) Design of an organic message of resource for the *OSOrderWithCustomerManagement* organic service exception return and of an organic message of resource for the *orderWithCustomerManagementCall* screen return.

The organic model in output of *osd2uml*, coming after the *fsd2osd* core transformation satisfies the rules specified in Section III.E. The organic components are represented in Fig. 7. Some rule illustrations are depicted below.

PresentationLayerComponentDesignRule means that the *PCOrderWithCustomerManagementScreen* component and its *orderCreationWithCustomerCall* operation result from the transformation of *UcOrderWithCustomerManagement* and its only scenario *ScOrderWithCustomerManagement*.

ServiceLayerComponentDesignRule indicates that the *OSCOrderWithCustomerManagement* component and its *OSOrderCreationWithCustomer* service result from the transformation of *UcOrderWithCustomerManagement* and its only scenario *ScOrderWithCustomerManagement*.

From **DataAccessLayerComponentDesignRule**, the *DACOrder* component results from the transformation of

LDOrder datum and its operation *createOrder* results from the transformation of the *createOrder* functional service provided by *FCOrderManagement*, which produces *LDOrder*.

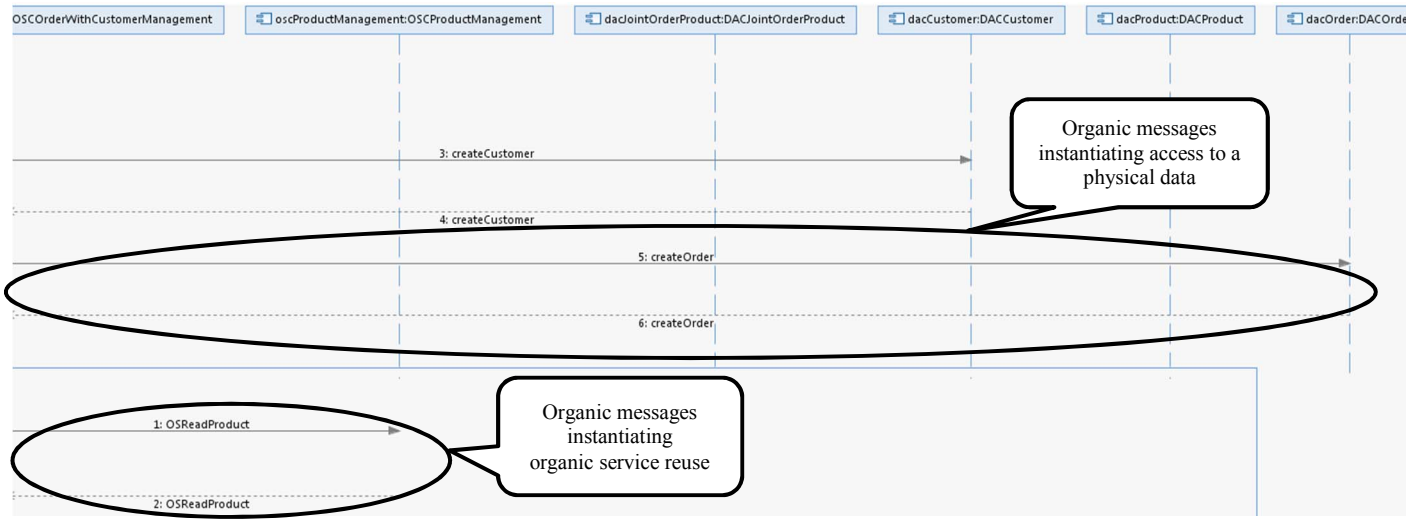

Fig. 6. Excerpt (see 4) of subsection IV.B) of the UML organic sequence diagram automatically generated.

EnterpriseArchitectureOrganicServiceReuseRule implies that the reuse of the *OSReadProduct* EA organic service during the *ScOrderWithCustomerManagement* scenario results in Fig. 7 from the instantiation of the *readProduct* functional service (see Fig. 2).

OrganicMessageDesignRule indicates that the *request and resource or*ganic messages instantiating the *createOrder* data access operation result respectively from the request and resource functional messages associated to the functional service named also *createOrder* (see Fig. 2).

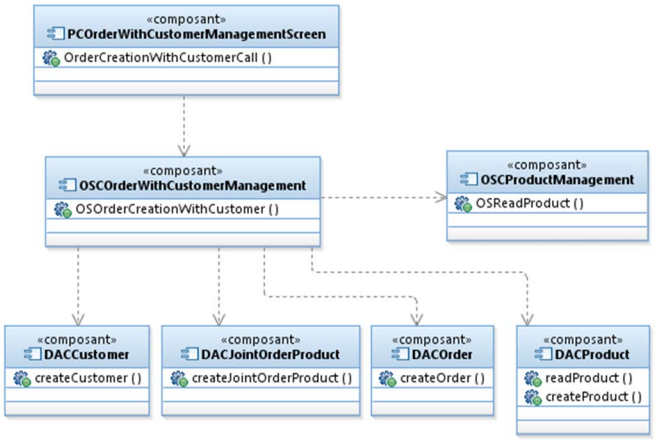

Fig. 7. UML representation of the organic component model automatically generated.

The physical data model is also automatically generated.

V. Discussion

In this section, the discussion focuses on the expected benefits of an automated design as defined above. An assessment is proposed with a gap analysis between this manual design and an automated design thanks to our transformation.

Table I represents a gap analysis between the automated organic architecture and the manual organic architecture deliveries. The automated organic architecture decreases the number of architecture mistakes thanks to the rules implementation. However, should we design the system architecture in relation to model transformations implementing these rules?

TABLE I. GAP ANALYSIS BETWEEN AUTOMATED AND MANUAL ORGANIC ARCHITECTURES

Manual organic architecture	Automated organic architecture
Tables to highlight the traceability	Rules implemented by the *fsd2osd* transformation
Eventual inconsistency of the temporal organic sequence with the functional sequence based on dependencies	Implemented consistency
Some mistakes for the processing in relation to a loop (sequence diagram and physical data model)	Implementation of the **OrganicMessageLoopDesignRule** enabling a specific processing of the temporal organic sequence

The use of the MDE approach targets the code generation utility for future programmers. The teaching experience depicted in [17] is for example a CRUD application context. The service layer is thus limited to the CRUD interface of the component of the data access layer. Unfortunately, system architecture design is more complex, especially in relation to the specification of system use scenario. MDE must help the system architect in order to address this complexity [18].

Nevertheless, a great difficulty for the architects is the adoption of a model based tooling. The tool comparison in [19] underlines this difficulty but only for code generation. In [20], the results show that EMF (Eclipse Modeling Framework) was

predominantly not easy to use by the developers in a case of conversion of a class diagram into a modeling language. Yet, EMF allows a better understanding of the models and this is a necessity to design a model to model transformation such as proposed in Section III.

The meta-modeling in relation to an organic architecture resulting from a functional architecture could be given and illustrated to assist the architects because the complexity of a meta-modeling supporting architecture rules. The project should be then the coding of simple architecture rules based on the meta-models' concepts. A simple architecture rule is for example **ServiceLayerComponentDesignRule** (see subsection III.E). This rule is implemented by the operational-QVT™ (imperative language for model transformation defined by the OMG [21]) code in Fig. 8.

```
//creation of a component of the service layer
mapping UseCase::createOrganicServiceComponent(oams : OAM) :
OrganicServiceComponent
{
          name := "OSC" + self.name.replace("Uc","");
          result.organicService := self.scenario->select(e : Scenario
| e.useCase.name=self.name)->asSequence()->map
createOrganicService(oams, result);
          result.oam := oams;
}
//creation of a service of a component of the //service layer
mapping Scenario::createOrganicService(oams : OAM, serviceComponent :
OrganicServiceComponent) : OrganicService
{
          name := "OS" + self.name.replace("Sc","");
          result.scenario := oams.scenario->select(e : scenario |
e.name=self.name)->asSequence()->first();
          result.oam := oams;
}
```

Fig. 8. Operational-QVT code of the **ServiceLayerComponentDesignRule**.

The coding of the simplest architecture rules could thus be another way to design system architecture.

VI. Conclusion and Perspective

Another experiment has been led with a most comprehensive CRM (Customer Relationship Management) system belonging to Commercial IS. This experiment shows the reuse of the services recommended by the EA of this IS and the use of services developed inside this CRM system.

All these experiments could support the teaching of system architecture based on MDE, including the coding of rules. A perspective of this work is the support of system architecture based on MDE for system architecture design. This perspective needs to design a learning entirely based on meta-modeling (definition and illustration of concepts useful to the system architects) and on rules based on this meta-modeling. The coding of these rules could be for these architects a new way to their activity: develop the architecture rules almost design the system architecture.

Finally, experiment this approach on an industrial use case is a big challenge.

References

[1] International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers, "ISO/IEC/IEEE 15288:2015 - System and software – System life cycle processes," 2015. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:15288:ed-1:v1:en. [Accessed 23 11 2016].

[2] International Organization for Standardization and International Electrotechnical Commission, "ISO/IEC 12207:2008 - System and software – Software life cycle processes," 2008. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:12207:ed-2:v1:en. [Accessed 23 11 2016].

[3] C. Haskins, K. Forsberg, M. Walden, D. Walden, and D. Hamelin, System engineering handbook, INCOSE, 2006.

[4] P. Kruchten, "The rational unified process: an introduction", Addison-Wesley Professional, 2004.

[5] J. Simonin, F. Alizon, J.-P. Deschrevel, Y. Le Traon, J.-M. Jézéquel, and B. Nicolas, "EA4UP: Enterprise Architecture-Assisted Telecom Service Developmennt Method," 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 279-285, 2008.

[6] J. A. Zachman, "Enterprise Architecture: The issue of the century," Database Programming and Design, 1997, pp. 44-53.

[7] D. M. Erikson, "A principal exposition of Jean-Louis Le Moigne's systemic theory," Cybernetics & Human Knowing, vol. 4, no 2-3, 1997, pp. 35-77.

[8] J. Long, "Relationships between common graphical representations in Systems Engineering", Vitech white paper, Vitech Corporation, Vienna, Austria, 2002, p. 70.

[9] Object Management Group, "Business Process Model and Notation (BPMN), version 2.0," 2011. Available: http://www.omg.org/spec/BPMN/2.0/. [Accessed 23 11 2016].

[10] OASIS, "Web Services Business Process Execution Language version 2.0", 2007. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. [Accessed 23 11 2016].

[11] Object Management Group, "Service oriented architecture Modeling Language (SoaML), version 1.0", 2012. Available: http://www.omg.org/spec/SoaML/. [Accessed 06 10 2016].

[12] A. Delgado, F. Ruiz, I. G.-R. De Guzman, and M. Piattini, "Business Process Service Oriented Methodology (BPSOM) with Service Generation in SoaML," International Conference on Advanced Information Systems Engineering, 2011, pp. 672-680.

[13] E. C. Salgado, J. Teixeira, N. Santos et R. J. Machado, "A SoaML approche for Derivation of a Process-Oriented Logical Architecture from Use Cases," International Conference on Exploring Services Science, 2015.

[14] P. Kruchten, H. Obbink et J. Stafford, "The Past, Present, and Future of Software Architecture," IEEE Software, vol. 23, no 2, 2006, pp. 22-30.

[15] Eclipse Foundation, "Eclipse Modeling Framework (EMF)," 2007. [Online]. Available: http://www.eclipse.org/modeling/emf/. [Accessed 02 11 2016].

[16] Object Management Group, "UML 2.2 Unified Modeling Language," 2009. Available: http://www.omg.org/spec/UML/2.2/. [Accessed 04 10 2016].

[17] J. Porubän, M. Bačíková, S. Chodarev, and M. Nosál', "Pragmatic model-driven software development from the viewpoint of a programmer: Teaching experience," IEEE Federated Conference on Computer Science and Information Systems (FedCSIS), 2014, pp. 1647-1656.

[18] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap". IEEE Future of Software Engineering, 2007, pp. 37-54.

[19] C. Eckert, B. Cham, J. Sun, and G. Dobbie, "From Design to Code: An Educational Approach," International Conference on Software Engineering & Knowledge Engineering, Knowledge Systems Institute, 2016.

[20] P. J. Clarke, Y. Wu, A. A. Allen, and T. M. King, "Experiences of teaching model-driven engineering in a software design course," Educators' Symposium of the MODELS Conference, 2009, pp. 6-14.

[21] Object Management Group, "MOF 2.0 Query / View / Transformation, OMG Technical Report," 2009. Available: http://www.omg.org/spec/QVT. [Accessed 02 11 2016].