A Role Language to Interpret Multi-Formalism System of Systems Models

Jean-Philippe Schneider*, Joël Champeau*, Ciprian Teodorov*, Eric Senn† and Loïc Lagadec*
*ENSTA Bretagne Lab-STICC CNRS UMR 6285, 2 rue François Verny, 29806 Brest Cedex 9, France
Email: {jean-philippe dot schneider, joel dot champeau, ciprian dot teodorov, loic dot lagadec} at ensta-bretagne dot fr
†Université de Bretagne Sud Lab-STICC CNRS UMR 6285, Rue de Saint-Maudé, BP 92116 56321 Lorient CEDEX, France
Email: eric dot senn at univ-ubs dot fr

***Abstract*—New systems are by nature distributed and built around existing systems. The concept of System of systems (SoS) has become the key for creating new systems. Modeling is a central issue of SoS design and evolution. Since subsystems of a SoS can be modeled in different languages, the SoS modeling environment should be able to handle heterogeneous modeling formalism. Traditionally, this problem is handled through type-based model transformations which are rigid and static. In this paper we address this problem by proposing a role modeling language offering a dynamic and extensible way to interpret and connect models from different languages. An usage of the role modeling language on a seafloor observatory use case is shown.**

I. Introduction

A system of systems (SoS) is a system in which subsystems are systems themselves. According to Maier's criteria [1], each subsystem 1) is functionally independent, 2) is managed independently, 3) is able to evolve at its own rate, 4) is able to cooperate with the other subsystems and 5) is geographically distributed. Due to the technical and management complexity of a SoS, Model-Based System Engineering is necessary to specify, design and analyze its structure and behavior. The management independence feature implies that each subsystem is modeled independently. Due to specific tooling and habits, different modeling languages are used according to the needs of the different subsystems [2]. The evolution feature of a SoS does not only encompass the system evolution but also the evolution of the design and analysis techniques. So, new modeling languages can be in use to design a subsystem and existing ones may become obsolete. The functional independence of the subsystems implies the need to get a global understanding of the different subsystems when building the SoS. A common way is to build a virtual system and to simulate it [3]. In this context, the issues of 1) "manipulating" homogeneously model elements of different modeling formalism, 2) insuring consistency between the conceptual and simulation models and 3) handling the evolution of modeling formalism over SoS lifecycle are raised.

To handle the multi-modeling languages issue, the current approaches rely on multi-viewpoints, stereotypes and/or a pivot meta-model [4]. However, these approaches imply static interpretation of models. To bypass this issue, we investigate the use of role modeling. Role models can be defined to interpret models in different formalism and provides a unified point of view on several models. In this paper, we present a dedicated modeling language to create role models and attach or detach the roles to model elements or to roles. Besides, the same entity (model element or role) can play multiple roles be they either different instances of the same role or instances of different roles.

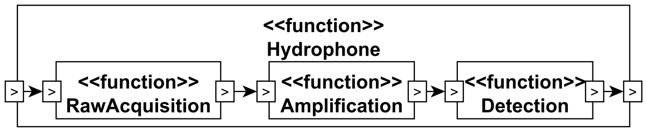

Fig. 1. Internal Block Diagram of an hydrophone

To assess our approach, we modeled a use case from a seafloor observatory. In this example, different modeling languages were used to model the subsystems. We used our role language to create a common representation of the seafloor observatory functional architecture starting from the models of the subsystems.

The rest of the paper is organized as follows. Section II raises the issues we faced on the seafloor observatory project. Section III provides background on the current approaches to handle the multi-modeling formalism issues and the role modeling. Section IV details the language we defined and the methodology we established. Section V illustrates the use of our language on our seafloor observatory use case. Section VI provides a conclusion and suggests further work.

II. Motivating Example

MeDON is a seafloor observatory SoS made of several sensors and a computing system aiming at detecting and localizing marine mammals through Passive Acoustic Monitoring [5]. The MeDON project [6] is an interesting example for the use of multiple modeling formalism.

In order to ease the design process and the communication between the experts of the different domains involved in the design, we adopted a Model-Based System Engineering (MBSE) approach mixing SysML [7] and eFFBD [8]. The eFFBD language is a flow-oriented modeling language to specify functional architectures. SysML was used to model the sensors. Fig. 1 shows a simplified Internal Block Diagram that describes a functional structure of an hydrophone. The block *Hydrophone* is made of functions to 1) acquire raw data (*RawAcquisition*), 2) amplify the signal (*Amplification*), 3) perform a detection algorithm (*Detection*). The eFFBD modeling language was used to capture a high-level view of the

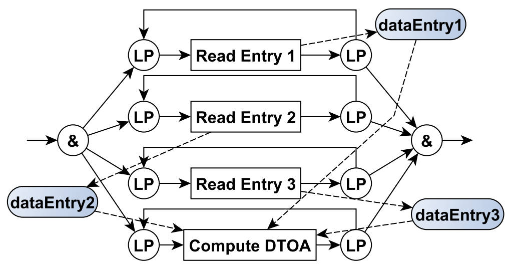

Fig. 2. eFFBD Diagram of the computing system

computing system, modeling the flow between the functions of the computing system. Fig. 2 shows the first level of the eFFBD model. The *ReadEntry1*, *ReadEntry2* and *ReadEntry3* functions receive data from sensors. They send a flow to *Compute DTOA* which stands for Difference of Time of Arrival that performs the localization of the sound source.

To better understand the system, a simulation strategy was applied at the functional architecture level. The use of two different modeling languages raised several issues when building a common simulable conceptual model:

- the models from the different stakeholders should be tested together. The complexity of the SoS and the management independence of each subsystem require to check the cooperation of the subsystems by simulating the SoS model;
- the same concept is modeled in different ways according to the used modeling language. The functions were modeled using *SysML blocks* and *eFFBD functions*. To obtain the simulable model the semantic gaps between languages need to be bridged;
- the models of the subsystems should be correctly instantiated in the SoS model. The observatory designers will have to integrate into the observatory model three instances of the modeled hydrophone defined in an external model in order to meet the requirements of the signal processing part of the observatory. To address this issue, the SoS modeling environment needs to provide tools to create instances of external models;
- during the lifecycle of the seafloor observatory, the modeling formalism will evolve. For example, between version 1.2 and version 1.3 of SysML the notion of port has changed. We need to have a mechanism with dynamic properties independent of a specific formalism.

This motivating example highlights the goals of 1) sharing a common semantics between several formalism, 2) producing coherent simulation results regarding the shared semantics, 3) having an instantiation mechanism between and through several formalism.

In this context we analyzed the current approaches and identified the major drawbacks related to the rigid definition of models which currently restricts the interoperability between the formalism. The next section presents this analysis on the modeling languages and introduces role modeling as an approach to face the rigidity in modeling language definitions.

III. Background

Although the type system in modeling languages eases the analysis of models, it provides an excessive rigidity to the extend that becomes an issue when the formalism interoperability is one of the key feature like in SoS's modeling. We now first present the major drawbacks of the current type modeling approaches. We then introduce the role modeling to bypass the rigidity of these approaches.

A. Type-based approaches

In a traditional modeling language, each model element is identified by a type providing a unique semantics. A type enables the classification of elements according to their essence i.e. what they are. Therefore, types guide the transformations among different modeling formalism so as to keep the meaning of the modeled elements. Viewpoints, stereotypes and pivot meta-model are three approaches that handle the heterogeneity of modeling formalism.

A viewpoint defines a concern to group model elements. It eases the understanding of system models by grouping the model elements focusing on the same concerns of the system. Architecture Frameworks such as the DoDAF [9] or the MoDAF [10] define multiple viewpoints. Only specific model elements can be used in a given viewpoint. The main drawback of this approach is the possibility to define multiple models of the same element. For example, an eFFBD function may be modeled once as a function in a behavioral point of view and once as a block in a structural point of view. Two model elements represent the same concrete entity with possible duplicate information. This may create inconsistencies in the model. For example if the concrete entity is renamed, all associated model elements are impacted. If no proper methodology is used, there is no guarantee that all the impacted elements will be modified.

Stereotypes are tags applied to model elements. Stereotypes enable to extend a modeling language as in the common DoDAF/MoDAF profile in SysML (UPDM) [11]. For example, an *eFFBD function* stereotype can be added to the SysML modeling language. However, model elements may suffer from *object schizophrenia*: once the stereotypes is applied, it becomes difficult to understand what the model element really represent.

A pivot meta-model is a modeling language able to define all the concepts of the modeling languages to be merged together. The different input models are transformed into a global model compliant to the common modeling formalism. This approach implies that the pivot meta-model is able to model all the expected aspects. For example, the modeling formalism will have to cover all the aspects of the design of a functional architecture in order to avoid losing information from the input models. As a result, The common formalism will become huge and complex. In addition, the model transformations have to be verified against inconsistencies between the result model and the input models. All the modeled features of the input models must be found in the output model even when they are not needed anymore. For example, the functional hierarchy modeled in the input models have to be present in the result model.

During the long lifecycle of a SoS, new modeling formalism may be used making it tedious to used the described approaches; the new concepts may require to define new viewpoints, or new stereotypes or even to extend the pivot meta-model. All these activities require to revisit all the transformations and the consistencies rules.

B. Role Modeling

Over the last decades, role modeling has been used at several levels (conceptual, meta-modeling, programming) to address the rigidity issue of the classical typed approaches [4]. Among the examples of role modeling usages are multi-level modeling, database querying and toolchain building. The notion of role is complementary to the notion of type. In contrast to a type, at the model level, a role is applied to an element according to its relationships with other elements [12]. Figure 3 illustrates an example of distinction between types and roles using a SysML and an eFFBD models. These two

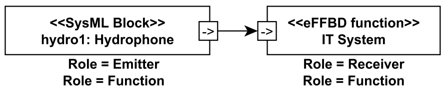

Fig. 3. Example of the distinction between types and roles. Types can be defined in multiple formalism. One instance can hold several roles.

blocks are linked together by a flow. In this relation, *hydro1* can be interpreted as a flow emitter and *IT System* can be interpreted as a flow receiver. The notions of flow emitter and flow receiver are both roles related to the type *SysML Block* and the type *eFFBD Function* at instance level. At the meta-model level, a mapping between types and roles can be defined to indicate the interpretation of the types according to the roles. Roles are dynamic. They can be attached to model elements and detached according to the context. The interpretation of the model elements can evolve over time.

Steimann lists a set of 15 features that may be fulfilled by a role-based modeling framework [12]. The framework designer should choose the features needed for his applications. Kühn and al. upgraded Steimann's features list to add the ability to group roles to define a context [4]. The benefit of these features was illustrated at both meta-modeling, modeling and implementation levels. Steimann cites usage of roles to dynamically define the meaning of association ends in an UML class diagram. He also cites the use of roles to give meaning to instances in an instantiation of a design pattern at code level.

Usage of roles can facilitate data management. Databases are able to store large pieces of structured information. However, databases are only able to capture the static features. Halpin [13] describes how roles can be used to capture the dynamic aspects of the data meaning. He defines a seven-step methodology demonstrating the flexibility brought by the usage of roles. In his experiment the static data is stored in a database and the roles are used to conceptually define queries on the set of data stored. The roles enable to create context-specific adaptation queries according to the relations that exist between data. The roles also bring the ability to modify the queries in case of a context switch.

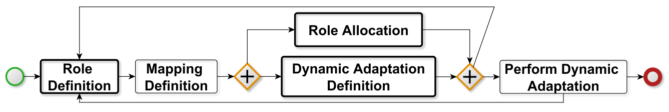

Fig. 4. BPMN diagram of our methodology

Roles can also be used to build toolchains. Currently, the design of a system or a software involves many tools. All these tools are complementary. However, making them work together is difficult as they are not built to this end. Seifert and al. [14] show that roles can be used to build the missing links so that models created in one modeling tool can be reused in a completely different one. Roles define how the model elements should be interpreted. Seifert and al. define a three-step approach to build a toolchain using roles. First, the role model is defined. Then, the different roles are attached to the model elements. Finally, attribute of the model elements are interpreted. This approach has the advantage of not modifying the tools meta-models. Unlike Seifert and al., we use role to be able to merge entities representing the same concept but modeled in different modeling languages.

Roles are successfully used in multi-tool environments and bring context information to strongly typed data. These properties make them interesting in a SoS context.

IV. Role4All: a Role-Based Framework

In this section, we introduce a role language aiming at providing tooling to interpret models expressed in different modeling languages. This language, named Role4All, is used to define and allocate roles. Before presenting the language and its use with a Smalltalk implementation, we introduce the process designed for Role4All.

A. Role4All Process

The BPMN diagram in Fig. 4 summarizes the different activities required to use roles.

To handle several models using roles, we first need to identify the role model linked to the context, in our case the functional model. The role model must contain all the roles, with attributes and behavior, representing the common interpretation between the formalism (this activity is described in section IV-D). One of the specificities of the role model is to be considered as dynamic information, so that it can be updated iteratively and independently of the formalism. For example, this role model can be extended or reduced based on the introduction of new formalism in our context. After the role definition, the mapping activity includes the specification of the mapping between roles and model elements. Again, this mapping is dynamic and can evolve over time to take into account new formalism. The mapping definition is described in the section IV-D.

The next steps can be processed in parallel (see Fig. 4) and are necessary to constitute a complete role definition. Nevertheless, the two activities are not mandatory to achieve a usable role definition in the modeling framework. The first activity is the allocation of the role instances to the model elements so as to strengthen the model element behavior

TABLE I. REQUESTED ROLE FEATURES

F1	Roles can be attached to elements of unrelated types
F2	roles can be played either by types or roles
F3	groups of roles can be defined
F4	the same object can play multiple roles
F5	a role has its own identity
F6	a role has its own behavior

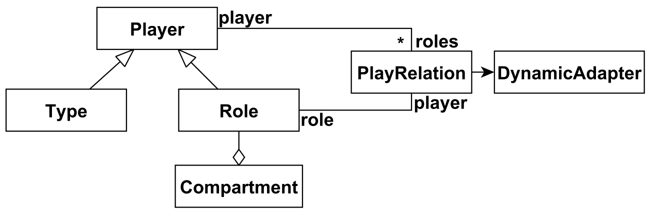

Fig. 5. Structural model of the role description language

with the role one. Each role instance is attached to model elements based on the mapping definition of the previous activity. The allocation activity is illustrated in Section IV-D. In parallel, we can associate to the roles the necessary definition of the adaptation of the model elements (defined through the relationships mapping) corresponding to the role behavior. This adaptation definition and usage is presented in Section IV-D.

The last activity is to perform the dynamic adaptation. It includes the creation of an interpreter chain between several models including the role models (Section IV-E).

B. Desirable Role Features

From [4] we extracted the role features required to handle multiple modeling formalism in a SoS context. These features are shown in table I. Feature F1 enables to handle multiple modeling formalism. A role *Function* can be attached to both *SysML Block* and *eFFBD Function* types. F2 enables to chain interpretation to extract information from models and then to reinterpret it in a new context. For our simulation use case, we define the role *AtomicModel*, a kind of discrete event simulation component [15], and attach it to the role *Function*. F3 enables to clearly state the context in which the roles are applied. We need to group together the roles defined to describe a functional architecture and to make another group for the roles defined for the simulation. Features F4, F5 and F6 enable to deal with the issue of creating multiple modeling elements compliant with the specification from the subsystems. We are able to virtually create multiple instances of the same modeled object. We create multiple instances of the same role and attach it to the same modeled element from the subsystem models. Each instance of the role has its own properties' values in order to be distinguished from the others instances of roles and its own behavior.

C. Role Language Specification

To fulfill all these features, we define the structure of our modeling language as shown Fig. 5. The *Type* class represents the mother class used as root in the meta-modeling languages (F1). For example, for the SysML language it can be the *Block* meta-class. The *PlayRelation* class reifies the allocation of a role to an instance of the *Player* class. The inheritance relationship between *Player*, *Type* and *Role* enables to allocate a role to both a type and a role (F2). The *Compartment* class enable to group roles (F3). The *0..n* multiplicity of the association between the *Player* and the *PlayRelation* classes enables an instance of *Player* to play multiple roles (different ones or the same multiple times)(F4) and the dynamic attachment/detachment of roles. This set of classes defines a role meta-model. Each role in the role model can be instantiated multiple times. Each instance is independent and has its own identity (F5). As roles are classes, they may hold attributes. The values of the attributes are different in each instance of a role. Roles also have associated methods to define their behavior (F6). We can distinguish two kinds of behavior. The first kind of behavior is related to the model element that plays the role. This behavior is adapted to the allocated type or role instance through an instance of the *DynamicAdapter* class. The second type of behavior is completely dedicated to the role.

D. Role Definition, Mapping and Allocation

An example of the definition of a role and of role's attributes addition in the Role4All framework is shown in Listing 1.

Listing 1. Definition of a role
```
Role named: 'Channel'.
Channel addAttributeNamed: 'isAsynchronous'.
```

Line 1 shows the creation of the role. The *Role* class implements the *Prototype* design pattern. A class named *Channel* is created by cloning the *Role* class. Line 2 shows the addition of an attribute to the *Channel* role. Attributes in roles are handled as a dictionary in the role classes. Each instance of a role class may have a different value for each attribute.

Listing 2 shows the definition of mappings.

Listing 2. Role mappings definition
```
Port plays: Channel.
p1 plays: Channel withIdentifier: 'channel1'.
p2 plays: Channel.
```

Line 1 shows the mapping between a type and a role by using the *plays* method defined at the class level. All the instances of type *Port* will play the *Channel* role. To attach an instance of *Role* to a specific instance of *Player*, we added the method *plays:withIdentifier:* to the *Player* class at the instance level as shown at Line 2. *p1* is an instance of a SysML port. For example the input port of the *Hydrophone* block shown in Fig. 1. Calling this method creates an instance of *Channel* role and associates the *channel1* identifier to the created instance. An instance of *PlayRelation* is created too. The player association of the instance of *PlayRelation* is set to *p1* and the role association is set to the created instance of *Channel*. Line 3, is similar to Line 2 except that it does not specify an identfier. In that case, the identifier is generated by the *Channel* class.

After creating and associating roles to model elements, it is mandatory to describe how the model element should be interpreted by the role as shown in Listing 3.

Listing 3. Definition of the dynamic adaptation of a *Player* by a *Role*
```
Channel linkedToKind: Port actsAs:{codeBlocks}
Channel identifiedBy: 'channel1' actsAs:{codeBlocks}.
```

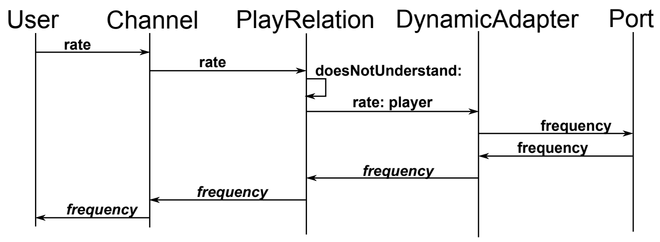

Fig. 6. Example of the delegation between a *PlayRelation* and a *DynamicAdapter*

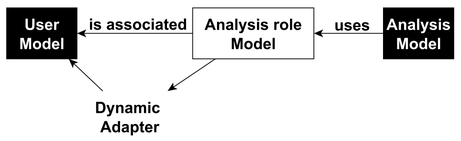

Fig. 7. Usage of our role model to create an interpreter chain

This interpretation by the role is either based on the type of the *Player* or on the specific instance of role. Line 1 highlights the definition of an interpretation of model elements of kind *Port* by all instance of *Channel* role. An instance of the *DynamicAdapter* class is created and linked to each instance of *PlayRelation* that associates an instance of *Port* and an instance of *Channel*. The argument called *actsAs* is a set of blocks that defines the concrete implementation of the interpretation. This set of blocks is attributed to the instance of *DynamicAdapter*. Line 2 shows how to define or redefine the interpretation of a model element in a specific case. The interpretation by *channel1* role instance is modified. The new interpretation behavior replace the generic rule defined Line 1.

The *PlayRelation* class should provide a method call delegation mechanism to redirect the calls from the instances of *Role* to the instance of *DynamicAdapter*. As shown in [16], we avoid the implementation of all the called methods in the *PlayRelation* class by overriding the method *doesNotUnderstand:*. Fig. 6 illustrates the principles of the delegation. In this example, the *User* wants to retrieve the *rate* of a *Channel*. As *Channel* is a role, the request is sent to the *PlayRelation* to retrieve the information from the model element associated to the role. This call to an unimplemented method triggers the *doesNotUnderstand* method of the *PlayRelation*. In the *doesNotUnderstand* method, the call is transferred to the *DynamicAdapter* associated to the *PlayRelation*. As the *DynamicAdapter* has no link with the model element, the method call contains a reference to the model element. The *DynamicAdapter* can call the right method to get the *rate* from the model element. In this case, *rate* is called *frequency* in the *Port* class. The *DynamicAdapter* makes the translation. The retrieved information is sent back to the user.

E. Performing Dynamic Adaptation

Based on the *DynamicAdapter* mechanisms, we illustrate in Fig. 7 the use of dynamic adaptation in our framework. System architects have one or more models in one or more modeling formalism. These models should be used as input of tools such as other modeling tools, analysis tools, simulation tools, etc. To make the transition between tools, users define a role model.

TABLE II. ROLE ASSOCIATION TO SYSML AND EFFBD TYPES

SysML type	eFFBD type	Role
Block with sub-blocks	And operator	LogicalFunctionGroup
Block without sub-blocks	Loop operator	LeafFunction
Link	Flow	CommunicationLink

The roles define the interpretations of the original models. The obtained role has attributes and methods, thus giving to the final tool the ability to interpret the original models. The methods of the role get the pieces of information stored in the original model elements. This operation is performed by an instance of *DynamicAdapter* as specified by the users. This instance is associated to the existing link between the model element and its attached role. The *DynamicAdapter* transforms the pieces of information into a format corresponding to the role. The final tool has access to the information contained in the model through the interface provided by the roles. The final tool does not have to care on the adaptation of the information. In our example, information is always usable by the simulation tool whatever is done on the original models.

V. Application to the Seafloor Observatory

Based on the seafloor observatory system of systems example, we define a role model to describe a functional architecture. Using our role modeling language, three roles were defined: *LogicalFunctionGroup*, *LeafFunction* and *CommunicationLink*. The role *LogicalFunctionGroup* describes a logical grouping of functions. The role *LeafFunction* describes the functions that hold a behavior. The role *CommunicationLink* describes the exchanges existing between two *LeafFunctions*.

We define a set of rules to associate the functional architecture roles to the SysML and eFFBD types in the context of our example. These rules are summarized in Table II. The SysML blocks can be interpreted either as *LogicalFunctionGroup* or as *LeafFunction* depending on their holding of sub-blocks. The *SysML links* between ports are interpreted as *CommunicationLink*. The *And operator* of the eFFBD language is interpreted as *LogicalFunctionGroup* as it holds several *eFFBD branches* containing other functions. In our example, the *Loop operator* of the eFFBD language is interpreted as *LeafFunction* as only one function is on the *eFFBD branch* held by the operator. As shown by this example, the interpretation of the different elements is highly dependent on the relations between the instances of the model elements.

A manual allocation of roles to model elements is not appropriate when enforcing generic rules and with large models. As a result, we implemented the rules shown in Table II with Parser Combinators. Parser Combinators enable to iterate over the models and to enforce the rule automatically. Listing 4 shows the implementation of the rules to deal with eFFBD models with Parser Combinators.

Listing 4. Using Parser Combinators to attach roles
```
[:instance | instance isAndOperator] ==>
  [:anAndOp | anAndOp plays:LogicalFunctionGroup.].
LogicalFunctionGroup linkedTo: AndOperator
  actsAs: {...}.
```

The syntax of the parser is [*rule*]==>[*action*]. The parser are matched against instances stored in models. The rule enables to test the nature of the instance. Each rule is associated to an action through the use of the ==> operator. The action will be triggered when the rule matches. The action may contain any code. Listing 4 details the implementation of the parsers needed to implement the allocations described in Table II. Line 1 implements the rules to check whether an instance into an eFFBD model is an *And Operator*. According to the matched rule it is possible to associate to the model element the correct *Role* and the correct interpretation code. This is done Lines 2, 3 and 4 with the call to the *plays:* and *linkedTo:actsAs* methods.

In the case of SysML models, the same role will be instantiated multiple times and associated to the same SysML Block. As this is not a generic action, we prefer performing the attachment of the roles statically. This is shown in Listing 5.

Listing 5. Attaching multiple instances of the same role to the same element

```
hydrophone plays: LFG withIdentifier: 'hydro1'.
hydrophone plays: LFG withIdentifier: 'hydro2'.
hydrophone plays: LFG withIdentifier: 'hydro3'.
```

Lines 1 to 3 show the creation of three instances of the role *LogicalFunctionGroup* (LFG) which are both associated to the same instance of *Block*. Similarly, it is possible to change the interpreter.

In this case study, our role language enabled to define an interpretation model for conceptual models made in both SysML and eFFBD modeling languages. We were able to make the interpretation model evolves according to our needs. We dynamically linked the roles with the elements in the SysML and eFFBD models. As the same roles were used for both models, we handled transparently the SysML and eFFBD models to create the simulation.

VI. Conclusion

Current approaches to handle models in multiple formalism are based on strong typing. They enable to enrich the static semantic of the modeled elements. However, they require a lot of work to accept newly used modeling languages and bring rigidity in the model interpretations.

In this paper, we presented a modeling language which addresses the integration of multi-formalism models in a SoS system through a role-based approach. The Role4All language is focused on the four issues we identified on the seafloor observatory example. Role4All provides a dynamic and extensible formalism to interpret, couple and instantiate heterogeneous models and to handle the evolution of modeling formalism. Role4All gives the SoS designers the possibility to perform global specialized simulation regardless of the modeling languages used for the subsystems.

To show the characteristics of our approach we have modeled a specialized role model to simulate a seafloor observatory system of systems which subsystems are specified through SysML and eFFBD models. This approach enabled to manipulate indistinctly the concepts used in SysML and eFFBD. We were able to modify the interpretation of the models dynamically focusing on the elements modified in the source models.

In the future, we plan to foster the complementarity of our approach with powerful simulation and analysis environments, such as OMNeT++ or MS4Me. Moreover, the dynamic features enable to cope with evolution of the SoS. The formalization of the Role4All language and its integration in a tool supported SoS design methodology will provide the SoS designer with an environment enabling agility in the SoS design process.

Acknowledgment

This work has been done with the financial support of the French Délégation Générale de l'Armement and the Région Bretagne.

The authors also wish to thank Bruno Aizier for the valuable exchanges of ideas around this work.

References

[1] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.

[2] J. Sztipanovits, T. Bapty, S. Neema, L. Howard, and E. Jackson, "Openmeta: A model-and component-based design tool chain for cyber-physical systems," in *From Programs to Systems. The Systems perspective in Computing*. Springer, 2014, pp. 235–248.

[3] B. P. Zeigler and H. S. Sarjoughian, *Guide to Modeling and Simulation of Systems of Systems*. Springer, 2013.

[4] T. Kühn, M. Leutäuser, S. Götz, C. Seidl, and U. Aßmann, "A metamodel family for role-based modeling and programming languages," in *Software Language Engineering*, ser. Lecture Notes in Computer Science, B. Combemale, D. Pearce, O. Barais, and J. Vinju, Eds. Springer International Publishing, 2014, vol. 8706, pp. 141–160. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11245-9_8

[5] W. M. Zimmer, *Passive acoustic monitoring of cetaceans*. Cambridge University Press, 2011.

[6] Interreg IVA, "Marine edata observatory network," 2013. [Online]. Available: http://medon.info/

[7] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*. Elsevier, 2011.

[8] J. Long, "Relationships between common graphical representations in systems engineering," *Vitech white paper, Vitech Corporation, Vienna, VA*, 2002.

[9] US Department of Defense, "Department of defense architecture framework," http://dodcio.defense.gov/TodayinCIO/DoDArchitectureFramework.aspx, 2010.

[10] UK Ministry of Defense, "Ministry of defense architecture framework," https://www.gov.uk/mod-architecture-framework, 2012.

[11] M. Hause, "Model-based system of systems engineering with updm," http://www.omg.org/ocsmp/Model-Based_System_of_Systems_Engineering_with_UPDM.pdf, 2010 (accessed October 3, 2014).

[12] F. Steimann, "On the representation of roles in object-oriented and conceptual modelling," *Data & Knowledge Engineering*, vol. 35, no. 1, pp. 83–106, 2000.

[13] T. Halpin, "Object-role modeling (orm/niam)," in *Handbook on architectures of information systems*. Springer, 2006, pp. 81–103.

[14] M. Seifert, C. Wende, and U. Aßmann, "Anticipating unanticipated tool interoperability using role models," in *Proceedings of the First International Workshop on Model-Driven Interoperability*. ACM, 2010, pp. 52–60.

[15] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[16] B. A. Tate, *Seven languages in seven weeks: a pragmatic guide to learning programming languages*. Pragmatic Bookshelf, 2010.