Implementing the Behavioral Semantics of Diagrammatic Languages by Co-simulation

Daniel-Cristian Crăciunean[1]
Computer Science and Electrical Engineering Department
Lucian Blaga University of Sibiu, Sibiu, Romania

***Abstract*—Due to the multidisciplinary nature of cyber-physical systems, it is impossible for an existing modeling language to be used effectively in all cases. For this reason, the development of domain-specific modeling languages is beginning to become an integral part of the modeling process. This diversification of modeling languages often implies the need to co-simulate subsystems in order to obtain the effect of a complete system. This paper presents how behavioral semantics of a diagrammatic DSML can be implemented by co-simulation. For the formal specification of the language we used mechanisms from the category theory. To specify behavioral semantics, we introduced the notion of behavioral rule as an aggregation between a graph transformation and a behavioral action. The paper also contains a relevant example and demonstrates that the implementation of behavioral semantics of a diagrammatic model can be achieved by co-simulating standalone FMUs associated to behavioral rules.**

Keywords—DSML; cyber-physical systems; behavioral semantics; standalone FMU; FMI; diagrammatic language

I. Introduction

In the context of moving the effort from writing code to writing models, the development of modeling tools, appropriate to the domain of modeling, becomes an essential factor for increasing the efficiency of the modeling process. The diagrammatic syntax of domain-specific modeling languages (DSML) seems to be the most accessible for all parties involved in the model specification, because it is intuitive and can provide support in all phases of model development, starting with the informal model and ending with the executable model [1,2].

Models specified with these DSMLs must, in turn, interact with other models specified in other languages. Often the models specified with these DSMLs assemble heterogeneous components, which must be modeled with other languages. All these components can be specified in various modeling languages. But there is a need for a specific language to assemble the system components into a workflow [3] and coordinate the behavior of these components. In our opinion, these interaction problems can be solved elegantly by co-simulation [4].

One of the main objectives of building a model is to study the behavior of a system in order to analyze and optimize the modeled system. Due to the complexity of the systems, classical optimization methods cannot be used and therefore must be replaced by methods based on simulation or genetic algorithms. To achieve these objectives the model will have to be executed by a simulator according to its behavioral rules to mimic the behavior of the system.

Complex systems such as Cyber-Physical Production Systems (CPPS) also have a high degree of heterogeneity and therefore involve components with different behaviors that cannot be efficiently specified in the same formalism. In these cases, we need a co-simulation environment that combines several simulators into one and that reproduces the behavior of the global system [5].

In order for these heterogeneous models to be coupled in the co-simulation process, they need to provide a common standardized interface. This interface is called Functional Mock-up Interface (FMI) [6] introduced in the European MODELISAR project, carried out in the period 2008-2011.

To achieve the goal of co-simulation, modeling tools must be able to generate co-simulation units with FMI interfaces, which are called Functional Mock-up Units (FMU). The orchestration of the components in order to obtain the behavior of the composite system is done by an orchestrator which is called master algorithm.

We believe that for the efficient implementation of a DSML, co-simulation mechanisms must be an integral part in the process of specifying and implementing a modeling tool. In this paper we present the methodology for specifying and implementing a DSML with FMU generation facility. To formalize the model, we use mechanisms from category theory. For co-simulation we used the INTO-CPS [7] tool chain. INTO-CPS is an EU-funded project that integrates a chain of tools for model-based CPS design and implementation by co-simulating components with an FMI-compatible interface.

In Section 2, we briefly specify the static metamodel of a diagrammatic model. In Section 3, we specify the behavioral syntax of the model and in Section 4, we deal with the semantic mapping of a model. In Section 5, we briefly present the mechanism for generating FMU components. Section 6 concludes the paper with original contributions and conclusions. All the mechanisms presented are exemplified with a simple model that was implemented on the ADOxx metamodeling platform.

II. The Static Model

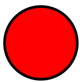
In essence, a visual model of a system first defines the syntax of the static and behavioral model that represents the virtual and physical entities of the model and then the

semantics of the model represented by the significance of static constructions and a set of behavioral rules that represent the behavior of these entities.

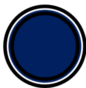
Syntactically, a diagrammatic model is a graph with several types of nodes that represent different concepts in the domain of modeling and several types of arcs that represent links between these concepts [8,9]. When we want to associate models with a spatial representation, we can use a second graph, as a spatial dimension of them and thus we reach the notion of bigraph [10]. The models discussed in this paper have as syntactic representation a single graph.

Example 1. We consider a modeling language SML (Simple Modeling Language) that has the following concepts:

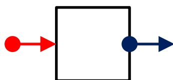
A buffer concept, which can store a single type of material, which we denote by B_1, and endow it with two attributes, namely: the stock attribute which represents the current quantity stored in the buffer and capacity which represents the maximum quantity that can be stored in the buffer. We associate to this concept the following graphic notation:



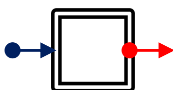
A buffer concept, which can store two types of materials, which we denote by B_2, and endow it with four attributes, namely: the attributes stock1, stock2 which represents the current quantity of each type stored in the buffer and capacity1, capacity2 which represents the maximum quantity of each type, which can be stored in the buffer. We associate to this concept the following graphic notation:



A processing or transfer activity concept, which we denote by W_1, and which can process or transfer materials from a type B_1 buffer to a type B_2 buffer, and endow it with three attributes, namely, the StockIn attribute which represents the quantity of material fed from buffer B_1, and the attributes Stock1Out, Stock2Out which represent the quantities of material of each type deposited in buffer B_2. We associate to this concept the following graphic notation:



A processing or transfer activity concept, which we denote by W_2, and which can process or transfer materials from a type B_2 buffer to a type B_1 buffer, and endow it with three attributes, namely: the attributes Stock1In, Stock2In which represents the quantities of material fed from each type and the StockOut attribute which represents the quantity of material deposited in buffer B_1. We associate to this concept the following graphic notation:



The SML model that we will specify is a graphical DSML for describing simple models in conformity with the requirements specified above.

SML models, therefore, are graphs with a set of syntactic restrictions on their components [11]. In the categorical model, the SML metamodel is a sketch that is composed of a graph and a set of constraints on the graph nodes [2,12,13].

Example 2. We will define a SML model as a graph $G=(X,\Gamma,\sigma,\theta)$, on the components of which we introduce four restrictions, namely:

1) The nodes of the graph are of two types and these types determine a partition on X, i.e.: $X=B1\sqcup B2$;

2) The arcs of the graph are of two types and these types determine a partition on Γ, i.e.: $\Gamma=W_1\sqcup W_2$;

3) Graph G has to be a connected graph;

4) There must be at most one arc between any two components.

A categorical sketch is a tuple $\mathcal{S}=(\mathcal{G},\mathcal{C}(\mathcal{G}))$ where $\mathcal{G}$ is a graph and $\mathcal{C}(\mathcal{G})$ is a set of constraints on the set of nodes and arcs of the graph [2]. A model of the sketch $\mathcal{S}=(\mathcal{G},\mathcal{C}(\mathcal{G}))$ is the image of this sketch through a functor in the Set category.

From the way of defining the SML model, from example 2 it results that the graph $\mathcal{G}$ of the corresponding sketch $\mathcal{S}=(\mathcal{G},\mathcal{C}(\mathcal{G}))$ is the one from Fig. 1.

The categorical sketches are based on the observation that a labeled diagram is an analogous construction of a logical formula that is mapped to the components of a graph, i.e. to the nodes and arcs of a graph [2,14].

We denote with Graph the category of graphs, i.e. the category that has graphs as objects and as arcs the homomorphisms between these graphs. We will also denote with $Graph_0$ the set of objects of the Graph category and with $Graph_1$ the set of arcs of the Graph category.

Constraints on the models specified by the categorical sketch are defined by a predicate signature diagram, which is composed of a set of predicates Π, and an application are: $\Pi\rightarrow Graph_0$, which maps the indeterminate predicates to the nodes of a graph in $Graph_0$ [2]. This predicate signature diagram, allows the definition of constraints on the models specified by a categorical sketch at the metamodel level.

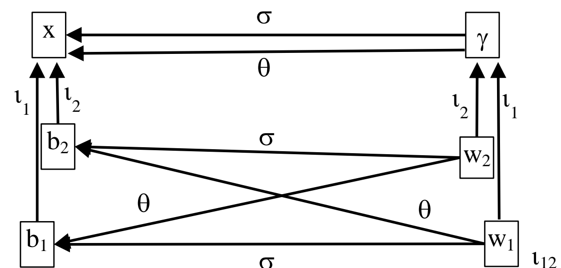


Fig. 1. The Graph of the SMM Sketch.

For example, for the graph G to be connected we will put the condition that the pushout of σ with θ to be a terminal object in the Set category.

If we denote Span(x,y,z,r_{zx},r_{zy})=($x \xleftarrow{r_{zx}} z \xrightarrow{r_{zy}} y$) then the pushout of σ with θ, in the Set category, is the colimit of the diagram d:Span(1,2,3,r_{31},r_{32})→Set where d(1)=x, d(2)=x, d(3)=γ, d(r_{31})=σ, d(a_{32})=θ. This constrains are imposed by the predicate P(n_1,n_2,n_3,r_{31},r_{32}) with the shape graph arity of P_3, ar(P(n_1,n_2,n_3,a_{31},a_{32}))=Span(1,2,3,r_{31},r_{32}) defined as: ar(n_1)=1, ar(n_2)=2, ar(n_3)=3, ar(a_{31})=r_{31}, ar(a_{32})=r_{32}. In these conditions the predicate P(n_1,n_2,n_3,a_{31},a_{32}) is defined as follows: P(n_1,n_2,n_3,a_{31},a_{32})=|CoLim(d)|=1 where CoLim(d) is the colimit of diagram d in the Set category.

Therefore, the categorical sketch of the SML model has the following components: the graph of the sketch is the one from Fig. 1, and the set of constraints $\mathcal{S}(\Pi)$ is obtained by mapping the shape graphs corresponding to the predicates from Π to the components of the sketch graph by means of diagrams, i.e. $\mathcal{S}(\Pi)=\{\mathcal{S}(P_i) \mid P_i \in \Pi, i \geq 1\}$. The categorical sketch $\mathcal{S} = (\mathcal{G}, (\Pi))$ represents the abstract syntax of the SML models and at the same time the SML metamodel.

Each model specified by the categorical sketch $\mathcal{S}$, is the image of the graph $\mathcal{G}$ of the sketch $\mathcal{S}$ through a functor M, in the Set category, which respects the constraints imposed by the predicates (Π). The predicates in the set (Π) will be mapped, at the level of each model M, from the Set category to a set of predicates as follows: Set(P_i)={(P_i;M∘d∘ar(P_i)) | d is a diagram}.

Thus, if we have the model M:$\mathcal{S}$→Sets, where M(b_1)=B_1, M(b_2)=B_2, M(w_1)=W_1 and M(w_2)=W_2, then the set of instances B_1, B_2, W_1, W_2, will respect the constraints defined by the set of predicates Set(P_i). We notice that the graph of the categorical sketch contains besides the concepts from the modeling domain, also auxiliary nodes useful for imposing constraints.

If in the above model we have: B_1={B_{11},B_{12},B_{13}}; B_2={B_{21},B_{22}}; W_1={W_{11},W_{12}}; W_2={W_{21}, W_{22},W_{23}} and σ(W_{11})=B_{11}; σ(W_{12})=B_{12}; θ(W_{11}) =B_{22}; θ(W_{12}) =B_{22}; σ(W_{21}) =B_{21}; σ(W_{22}) =B_{21}; σ(W_{23}) =B_{22}; θ(W_{21}) =B_{11}; θ(W_{22}) =B_{12}; θ(W_{23}) =B_{13} , then the SML model is like in Fig. 2.

We will consider that the nodes of the graph of the sketch $\mathcal{S}$ are classes endowed with attributes. The graph nodes will be mapped by the functor M to sets of objects of the corresponding class type in the Set category, and the graph arcs will be mapped to functions between these sets. The semantics of such a static model is given by the significance of the attributes, the significance of the values of these attributes and the significance of the graph structure of the model.

A class defines a concrete modeling concept that can be used to specify a model in the modeling language. Therefore, each concrete concept of a model created with a tool implemented on the ADOxx platform is an instance of a class. Each concrete class has a distinct name.

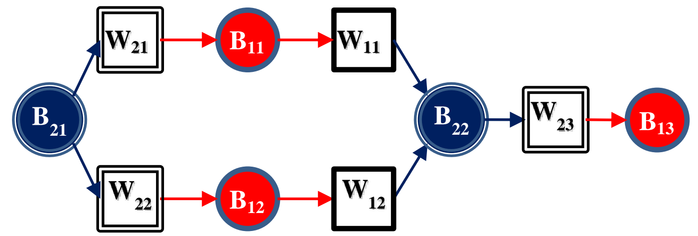


Fig. 2. Model Example.

III. Behavioral Syntax of the Modeling Language

Model transformation is one of the key techniques in MDE used especially for automating model management operations, such as code generation, model optimization, translation from one DSML to another, simulation, etc.

The transformation of diagrammatic models is based on, most often, the graph transformations defined by graph rules, also called productions. Such a production is a tuple p=(L, R), consisting of two graphs; a left graph L, a right graph R and a mechanism that specifies the conditions and how to replace L with R.

In this paper we will use graph transformations to model the behavior of a diagrammatic model. Graph transformation rules are mechanisms that can express the local changes of a graph in successive transformation steps ordered by a relationship of causal dependence of actions and therefore can accurately define the behavior of a diagrammatic model.

In the approaches of implementing the transformations, of the left graph to the right graph, two distinct mechanisms are distinguished, namely, the double pushout (DPO) and single pushout (SPO) [15,16]. Graph transformations allow the simultaneous transformation of the structure of the diagrammatic model and of the attributes of the components. In this paper we will use the DPO variant to specify the behavioral dimension of a model.

The correct application of a production p is made under the conditions in which the squares in Fig. 3 are pushout squares. A set of productions related to each other form a graph transformation system and can be used in the process of transforming models without being integrated into a graph grammar [15,17,18].

The behavior of SML models is not based on structural transformations of the graph but only on changing attribute values and therefore we will use graph transformations only to specify the context necessary to locate the components involved in a behavioral rule and to locate critical regions in the simulation process.

We will define the graph transformations at the metamodel level and they will be applied for any static model specified by the sketch in the Set category.

In the case of our SML model we have two transformations at the level of the sketch $\mathcal{S}$ from Fig. 1, namely, p1 (Fig. 4) and p2 (Fig. 5).

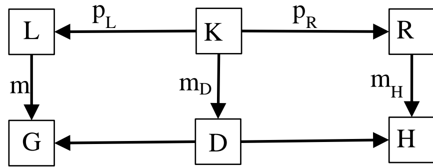


Fig. 3. A Double-Pushout Production.

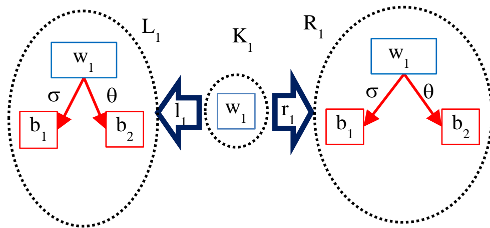


Fig. 4. Graph Transformation p_1.

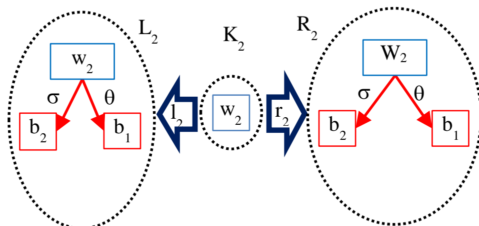


Fig. 5. Graph Transformation p_2.

IV. Semantics of the Modeling Language

A diagrammatic model is characterized by two dimensions: a static dimension, specified syntactically by the categorical sketch, and a behavioral dimension, specified syntactically by behavioral signatures. The two dimensions have dependent but still distinct semantics. Defining the semantics of the static dimension involves mapping the attributes with which the components of the graph of the categorical sketch are endowed to data domains and the graph structures of the model to well defined semantically structures, and defining behavioral semantics involves mapping the behavioral signature to mathematical functions.

The semantics of a model involves mapping attributes to their set of values, interpreting the graph structure of the model, while behavioral semantics highlights the structural and value transformations of the model.

The behavioral dimension of a SML model is defined by states and transitions. The states of the model are represented, in our approach, by the static models of the categorical sketch, and the transitions are represented by a set of mathematical functions associated with the behavioral rules. A behavioral rule can be applied only in the context in which a set of conditions are realized, represented by logical predicates that verify the state of the model. Therefore, a behavioral rule is defined as an association between a graph transformation and an action.

We will define the behavioral rules at the metamodel level by behavioral signatures represented by predefined actions with formal parameters, which will be evaluated at the moment of execution of a model, when they will receive current parameters corresponding to the respective concrete model.

Let the sets $Y_1,...,Y_n$, $U_1,...,U_m$. Then an action is an application $Act:U_1\times...\times U_m\rightarrow Y_1\times\ldots\times Y_n$ defined as follows: $(y_1,...,y_m):=Act(u_1,...,u_n)=(\omega_1(u_1,...,u_n), \ldots,\omega_m(u_1,...,u_n))$ where ω_i is an operation that calculates the value of y_i, i=1,m depending on the values of the variables $u_1,...,u_n$.

We denote with AGraph the category of graphs with attributes, i.e. the category that has as objects graphs with attributes and as arcs the homomophisms between these graphs. Also, if $G\in AGraph_0$, we will denote with attr(G) the set of attributes associated with the nodes and arcs of the graph G.

A diagram actions signature is a tuple $\Delta=(\boldsymbol{\mathcal{A}}, ar)$ where $\boldsymbol{\mathcal{A}}$ is a set of actions and ar is a function $ar:\boldsymbol{\mathcal{A}}\rightarrow Graph_0$ which maps each action $Act\in\boldsymbol{\mathcal{A}}$ to two objects in the AGraph category as follows: if $(y_1,...,y_m):=Act(u_1,...,u_n)$ then the outputs $y_1,...,y_m$ will be mapped to the attributes of the graph R and the inputs $u_1, ...,u_n$ will be mapped to the attributes of the graph L. The pair (L, R) of graphs is called shape graph arity of Act, ar(Act)=(L, R). We will sometimes denote the image of Act through ar in the category AGraph with Act(L, R).

The behavioral signature is a tuple $\Sigma=(\boldsymbol{\mathcal{T}},C_L,\Delta,C_R)$ where $\boldsymbol{\mathcal{T}}$ is a set of graph transformation rules; $C_L=(\Pi_L,ar_L)$ is a diagram predicate signature such that $ar_L:\Pi_L\rightarrow AGraph_0$, which we call the precondition signature; $C_R=(\Pi_R,ar_R)$ is a diagram predicate signature such that $ar_R:\Pi_R\rightarrow AGraph_0$, which we call the postcondition signature and Δ is a diagram actions signature, with the property that for any $Act\in\Delta$ there is $p\in\boldsymbol{\mathcal{T}}$, $p=L\overset{l}{\leftarrow}K\overset{r}{\rightarrow}R$ with ar(Act)=(L, R), that specifies how to transform the attributes of graph L which is the domain of action into the components of graph R which composes the codomain of the action.

We now denote the graph in Fig. 7 with $G_1(x_1, x_2, x_3)$ and the graph with a single node in Fig. 8 we denote it with $G_2(x_1)$. The shape graphs represent the local structures of a model and represent the areas of action of the behavioral rules in the context of a concrete model.

These shape graphs represent the local structure of the model, and the context in which a behavioral rule evolves. Behavioral signatures defined on the components of these shape graphs are mapped to behavioral transformations on the component elements of a model.

The behavior of the SML model can be specified by a behavioral signature that contains two behavioral rule signatures $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$. Since the set of behavioral rule signatures is equivalent to the behavioral signature, we will use the same notation $\boldsymbol{\Sigma}$ for the set of behavioral rule signatures.

So $\Sigma=\{\sigma_1,\sigma_2\}$ where:

$\sigma_1=(L^1\overset{l_1}{\leftarrow}K^1\overset{r_1}{\rightarrow}R^1,C_L^1,Act^1,C_R^1)$; $\sigma_2=(L^2\overset{l_2}{\leftarrow}K^2\overset{r_2}{\rightarrow}R^2,C_L^2,Act^2, C_R^2)$;

$L_1 = R_1 = L_2 = R_2 = G_1(1,2,3)$; $K_1 = K_2 = G_2(1)$;

$C_L^1=(\Pi_L^1, ar_L^1);\Pi_L^1=\{P_L^1(u_1,\ldots,u_n)\};ar_L^1(u_i)=a_i$, i=1,n and $a_i \in attr(L^1)$;

$C_R^1=(\Pi_R^1,ar_R^1);\Pi_R^1=\{P_R^1(y_1,\ldots,y_m)\};ar_L^1(u_i)=b_i$, i=1,m and $b_i \in attr(R^1)$;

$L_2= R_2=G_1(1,2,3)$; $K_2 = G_2(1)$;

$C_L^2=(\Pi_L^2 , ar_L^2)$; $\Pi_L^2=\{P_L^2(u_1,\ldots,u_n)\};ar_L^1(u_i)=a_i$, i=1,n and $a_i \in attr(L^2)$;

$C_R^2=(\Pi_R^2, ar_R^2)$; $\Pi_R^2 =\{ P_R^2(y_1,\ldots,y_m)\};ar_L^1(u_i)=b_i$, i=1,m and $b_i \in attr(R^2)$;

The behavioral signatures thus defined will be transformed into behavioral rules at the level of the metamodel, by mapping them to the components of the sketch $\boldsymbol{\mathcal{S}}$, and will represent the behavioral model at the level of the metamodel, i.e. the abstract behavioral semantics of the models. The behavioral rules at the level of the sketch $\boldsymbol{\mathcal{S}}$ will then be mapped by matches at the level of the models.

The behavioral rule signatures must be mapped to the components of the graph $\boldsymbol{\mathcal{G}}$ of the sketch $\boldsymbol{\mathcal{S}}$ by sets of three diagrams, one for each of the graph forms L, R and K. These will be defined by three functors d_L, d_K and d_R, where d_K is the restriction of the functors d_L and d_R at domain K; $d_k=d_l/K=d_r/K$, l_s and r_s are monomorphisms that inject the graph K into L and R, respectively.

We will therefore define the diagrams corresponding to the signature of the behavioral rule σ_1:

$d_l^1:G_1(1,2,3)\rightarrow G_1(\gamma_{12}, w_1, w_2)$ defined as $d_l^1(1)=\gamma_{12}$; $d_l^1(2)=w_1$; $d_l^1(3)=w_2$;

$d_r^1=d_l^1$; and $d_k^1:G_2(1)\rightarrow G_2(\gamma_{12})$ defined as restriction $d_k^1=d_l^1/K^1$; $d_k^1(1)=\gamma_{12}$.

And for the signature of rule p_2 we have the diagrams:

$d_l^2:G_1(1,2,3)\rightarrow G_1(\gamma_{21}, w_2, w_1)$ defined as $d_l^2(1)=\gamma_{21}$; $d_l^2(2)=w_2$; $d_l^2(3)=w_1$;

$d_r^2=d_l^2$; and $d_k^2:G_2(1)\rightarrow G_2(\gamma_{21})$ defined as restriction $d_k^2=d_l^2/K^2$; $d_k^1(1)= \gamma_{21}$.

Diagrams are functors that map the formal parameters defined by graph shapes to the concepts specified by the nodes of the sketch graph. The same graph shapes are, on the other hand, mapped to the components of a concrete model through matching applications.

A behavioral rule of the sketch $\boldsymbol{\mathcal{S}}$ is a tuple $t=(L\overset{l}{\leftarrow}K\overset{r}{\rightarrow}R, d_L(C_L),Act(d_L(L);d_R(R)),d_R(C_R))$ where $\sigma=(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R,C_L, Act(L;R),C_R)$ is a signature of a behavioral rule. We used the following notations: $d_L(C_L)=(\Pi_L,d_L(ar_L))$; $d_R(C_R)=(\Pi_R,d_R(ar_R))$.

Thus, starting from a behavioral signature, we generate a set of behavioral rules at the level of the metamodel, i.e. at the level of the components of the graph of the sketch.

If we denote with (Σ), the set of behavioral rules induced by the behavioral signature Σ, then a behavioral metamodel is a tuple $(\boldsymbol{\mathcal{G}}, \boldsymbol{\mathcal{S}}(\Sigma))$ where $\boldsymbol{\mathcal{G}}$ is the graph of the sketch $\boldsymbol{\mathcal{S}}=(\boldsymbol{\mathcal{G}}, \boldsymbol{\mathcal{C}}(\boldsymbol{\mathcal{G}}))$.

In our approach each of these behavioral rules will be implemented as an FMU component. The behavioral metamodel corresponding to the SML language is defined by two behavioral rules $(\Sigma)=\{\boldsymbol{\mathcal{S}}(\sigma_1), \boldsymbol{\mathcal{S}}(\sigma_2)\}$ where:

$(\sigma_1)=(L^1\overset{l_1}{\leftarrow}K^1\overset{1}{\rightarrow}R^1,\{P_L^1(d_l^1(L^1))\},Act^1(d_l^1(L^1);d_r^1(R^1)), P_R^1(d_r^1(R^1)))$;

$(\sigma_2)=(L^2\overset{l_2}{\leftarrow}K^2\overset{r_2}{\rightarrow}R^2,\{P_L^2(d_l^2(L^2))\},Act^2(d_l^2(L^2);d_r^2(R^2)), P_R^2(d_r^2(R^2)))$;

Thus, for our SML metamodel we will implement two FMU components corresponding to the two behavioral rules (σ_1) and (σ_2).

In order for the behavioral rules specified in the metamodel (Σ) to be applied at the level of a concrete model we will have to find the matches of each behavioral rule from (Σ) in a model from $Mod(\boldsymbol{\mathcal{S}},Set)$.

A match of a graph $\boldsymbol{\mathcal{G}}=(N, A, s, t)$ in the image of a functor $\phi:\boldsymbol{\mathcal{G}}\rightarrow Set$ is a total monomorphism of graphs $m:\boldsymbol{\mathcal{G}}\rightarrow\phi(\boldsymbol{\mathcal{G}})$ which maps the graph $\boldsymbol{\mathcal{G}}$ to the graph $\boldsymbol{\mathcal{G}}_m=(m(N), m(A), m(s), m(t))$ so that $\forall y_i \in m(N) \Rightarrow \exists\ x_i \in N$ with $y_i \in \phi(x_i)$ and $\forall a_i \in m(A) \Rightarrow \exists\ r_i \in N$ with $a_i \in \phi(r_i)$ respecting the conditions of homomorphism $m(s(r_i))=m(s)(m(ri))$ and $m(t(r_i))=m(t)(m(ri))$ for all $r_i \in A$. We will denote the set of graph matches $\boldsymbol{\mathcal{G}}$ in $\phi(\boldsymbol{\mathcal{G}})$ with $m(\phi,\boldsymbol{\mathcal{G}})$.

In this way the graph transformations and the actions on the attributes will be executed on a concrete model.

Under these conditions, a behavioral model, in the Set category, contains all the behavioral rules induced by the behavioral signature Σ, in the Set category. We notice that the set of behavioral rules is specific to each concrete model, but they can be implemented generically at the metamodel level.

As we can see each behavioral rule τ in Set, defines an application $\tau:M_L\rightarrow M_R$, where $M_L,M_R:\boldsymbol{\mathcal{S}}\rightarrow Set$ are functors which represents the domain and codomain of the rule τ and all these behavioral applications together, maps the set $\boldsymbol{\mathcal{S}}(\Sigma)$ of behavioral rules of the sketch into a set of behavioral rules $Set(\Sigma)$ in Set.

In the case of the SML language the atomic behavioral rules in Set, τ are of the form $\tau=(t, \mu, M_L, M_R) \in \boldsymbol{\mathcal{S}}(\Sigma)$ where $t \in \boldsymbol{\mathcal{S}}(\Sigma)$ is a behavioral rule $t=(L\overset{l}{\leftarrow}K\overset{r_s}{\rightarrow}R, d_L(C_L),Act(d_L(L); d_R(R)), d_R(C_R))$, M_L, $M_R:\boldsymbol{\mathcal{S}}\rightarrow Mod(\boldsymbol{\mathcal{S}}, Set)$ there are two functors, and μ is a tuple of match $\mu=(m_L,m_K,m_R)$, $m_L \in m(M_L,L)$, $m_R \in m(M_R,R)$, and m_K is the restriction of m_L to K, so that the diagram in Fig. 6 is a double pushout.

For the model from Fig. 2 we have 5 behavioral rules in $Set(\Sigma)$, two for $\boldsymbol{\mathcal{S}}(\sigma_1)$ and three for $\boldsymbol{\mathcal{S}}(\sigma_2)$: $Set(\Sigma)=\{\tau_{11}, \tau_{12}, \tau_{21}, \tau_{22}, \tau_{23}\}$.

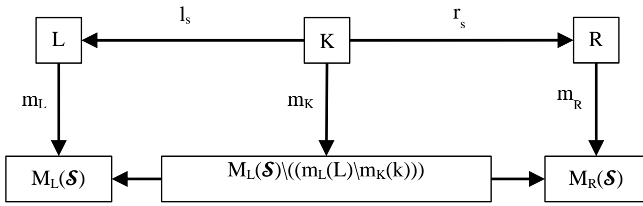


Fig. 6. A Double-Pushout Diagram.

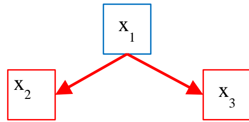


Fig. 7. Graph $G_1(x_1, x_2, x_3)$.

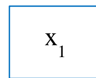


Fig. 8. Graph $G_2(x_1)$.

In our approach, behavioral rules are approached in two distinct phases. In the first phase, these rules are defined, at the metamodel level, by behavioral signatures, and in the second phase, these rules are applied at the level of each concrete model. If the behavioral rules of the model are faithful to the modeled system, then their successive application mimics the behavior of the modeled system.

V. Implementation of the FMU Generator

From the formalization presented in this paper results the fact that a diagrammatic metamodel has two dimensions, a static dimension represented by the categorical sketch and a behavioral dimension represented by the behavioral rules. A behavioral rule, as we have defined it, is an aggregation between a graph transformation on the structure of a model and a local action on the attributes of the model. If the functionalities of a metamodeling platform are designed to specify the graph structure of a metamodel and can be endowed with graph transformation facilities, behavioral actions are often performed by complex systems with a high degree of heterogeneity which implies the need for modeling on various modeling platforms. A solution to this problem is the assembly through co-simulation of n+1 independent components where n is the number of behavioral rules. In other words, you can build an FMU component that manages the static dimension of the model together with the graph transformations on it and an FMU component for each behavioral rule that models the action corresponding to the behavioral rule.

Applying a transformation rule specified by a behavioral signature $\sigma=(L\overset{l_s}{\leftarrow}K\overset{r_s}{\rightarrow}R,C_L,Act,C_R)$ is done as follows:

1) We first consider the diagrams d_L and d_R which maps the behavioral signature σ to the model sketch. In this way the components of the diagrams receive the types of components of the sketch.

When we are going to apply a behavioral rule $\tau=(t,\mu,M^L,M^R)\in\mathcal{S}(\Sigma)$ we have the first component M_L, which represents the current state of the behavioral model, and we are going to determine the M_R component which represents the state in which the transition is made. Therefore we can find the matches $m_L\in m(M_L,L)$ and $m_K\in m(M_K,K)=m(M_L,L)$ which are the first two components of a match. $\mu=(m_L,m_K,m_R)$ (Fig. 11).

2) The preconditions are verified, i.e. the fulfillment of the predicates defined by the C_L signatures, among which is the gluing condition. If the C_L conditions are met the graph transformation defined by the cospan $L\overset{l}{\leftarrow}K\overset{r}{\rightarrow}R$ is executed in two steps, 1 and 2.

a) We calculate the complement $M_L(\mathcal{S})\backslash((m_L(L)\backslash m_K(k))$ of the pushout of l_s with m_k, from Fig. 9.

b) Now we can calculate the pushout of r with m_k, from Fig. 10 and therefore the functor M_R and the component $m_R\in m(M_R,R)$ of the match μ.

All these transformations are executed temporarily, i.e. with the possibility of being canceled.

3) In this phase, the Act action is temporarily executed.

4) If the postconditions are also verified then the transformations described at points 2 and 3 are permanently executed, otherwise they are canceled by a rollback operation.

Obviously, independent behavioral rules can be applied simultaneously, and also the same behavioral rule can be applied simultaneously to several areas of the model if these areas do not contain common elements.

The model was implemented on the ADOxx metamodeling platform (see Fig. 12). In the case of the SML metamodel we defined, as it results from the analysis of the graph $\mathcal{G}$ of the sketch $\mathcal{S}$, two classes, corresponding to nodes b_1 and b_2 and two classes' relations corresponding to nodes w_1 and w_2. Defining classes in ADOxx is done visually, but the metamodel can be exported in ADL language or XML format. We used the ADL language to generate from classes, C structure types for standalone FMU.

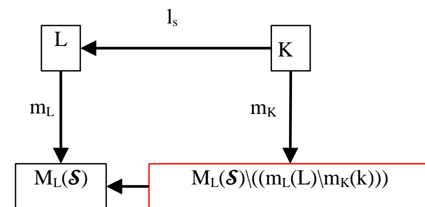


Fig. 9. The Complement of a Pushout.

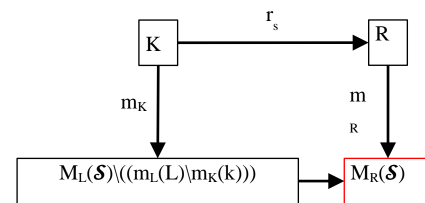


Fig. 10. A Pushout Diagram.

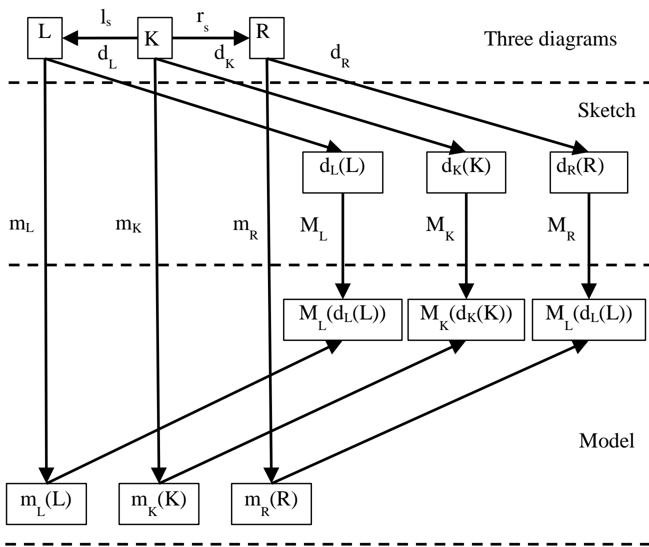


Fig. 11. Matching Three Diagrams.

The ADOxx metamodeling platform does not include behavior in classes and therefore we generated these classes as types of C structures. If the metamodeling platform would also include behavior this problem can be solved by function pointers. These structures were generated in a <ModelName>_fmu_types.h file, in our case SML_fmu_types.h. This type file is easy to write even manually, because it is written once and can then be used for all models specified with the implemented modeling tool.

We exported from ADOxx the model in ADL format from which we generated the FMU component corresponding to the static model, i.e. a graph structure corresponding to the specified model and with nodes that have corresponding types from the file <ModelName>_fmu_types.h and the name specified in the model. We put this graph structure in a file named <ModelName>_fmu_structure.h, in the case of the SML metamodel, SML_FMU_structure.h. Generating this graph structure is important because it is used in all specified models with the implemented modeling tool.

The behavioral part of the FMU component that manages the static dimension of the model was implemented manually in the C language. This is acceptable because it does not have a high complexity and is written only once for a modeling tool. The generation of this C code is possible but a translator from the ADOScript language to C should be implemented. To write this code we used FMU SDK [19] which can also be used in the case of generation from ADOScript.

FMU components corresponding to behavioral rules are usually written in another modeling tool. In the case of our SML metamodel, we specified the two components corresponding to the behavioral rules T_1 and T_2 in the VDM-RT language and exported them as standalone FMU.

Therefore, for the SML metamodel, we have 3 FMU components (Fig. 13) Which we briefly describe using the notations from [20]. In this sense an FMU is defined as a tuple $T_1=<S_c,U_c,Y_c,set_c,get_c,doStep_c>$; where: S_c is the space of states; U_c is the set of input variables; Y_c, is the set of output variables; $set_c:S_c\times U_c\times\nu\rightarrow S_c$ and $get_c:S_c\times Y_c\rightarrow\nu$ are the input and output functions and $doStep_c:S_c\times R_+\rightarrow S_c$ is a function that calculates the state after a given step.

$T_1=<S_1 ,U_1 ,Y_1 ,set_1 ,get_1 ,doStep_1>$;

$S_1=N\times x|x\in N,0\leq x\leq capacity1\}\times\{(x,y)|x,y\in N,0\leq x\leq capacity1 \text{ and } 0\leq y\leq capacity2\}$; $U_1=T1u=\{$ initial_id, initial_stock, initial_stock1, initial_stock2}; $Y_1=T1y=\{$ final_id, final_stock, final_stock1, final_stock2}; The parameters PT_1 of the component T_1 are: $PT_1 =\{$ capacity, capacity1, capacity2, stockIn, stock1Out, stock2Out }.

The $doStep_1$ function implements only the action Act^1 because we do not have structural transformations of the model. The precondition for the execution of the action Act^1 is: initial_stock≥stockIn and capacity1-initial_stock1≥stock1Out and capacity2-stock2≥stock2Out.

The action (final_stock, final_stock1, final_stock2)= Act1(final_stock, final_stock1, final_stock2) defines the operations; final_stock=initial_stock-stockIn; final_stock1= initial_stock1+stock1Out; final_Stock=initial_Stock2+ stock2Out. We will consider that we do not have postconditions in the case of the SML model.

$T_2=<S_2 ,U_2 ,Y_2 ,set_2 ,get_2 ,doStep_2>$;

In the case of SML: $S_2=S_1$, $U_2=U_1$, $Y_2=Y_1$ and $PT_2=PT_1$.

The $doStep_2$ function implements the Act^2 action as follows: The precondition for the execution of the Act^2 action is initial_stock1≥stock1In AND initial_stock2≥stock2In AND capacity-initial_stock≥stockOut. The action (final_stock, final_stock1, final_stock2)=Act^2(final_stock, final_stock1, final_stock2) defines the operations: final_stock1= initial_stock1-initial_stock1In; final_stock2=initial_stock2-initial_stock2In; final_stock=initial_stock+initial_stockOut. Even in the case of this behavioral rule we do not have a postcondition.

For component M we have the inputs and outputs identical to the inputs and outputs of the other two components in the case of the SML model. The $doStep_M$ function finds the matches in the model and implements the distribution of activities to the T_1 and T_2 components. Of course, for the T_1 and T_2 components there will be several instances, one for each match. In the case of the example in Fig. 2 we have 2 instances of the T_1 component and three instances of the T_2 component. The distinction between the two types of instances is made by the value of the variables initial_id and final_id. For the co-simulation of the three components we used INTO-CPS. We performed the co-simulation on an example of data and we obtained the output from Fig. 14. The graphs show the stocks resulting from the two instances of the T_1 component and three instances of the T_2 component.

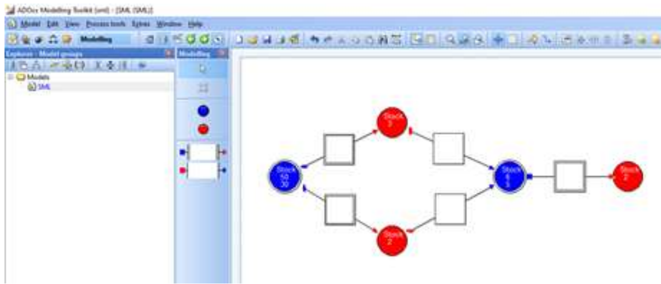


Fig. 12. SML Tool.

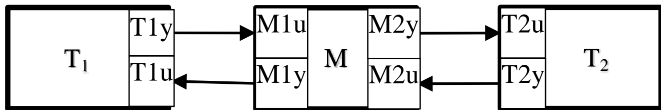


Fig. 13. FMU Components.

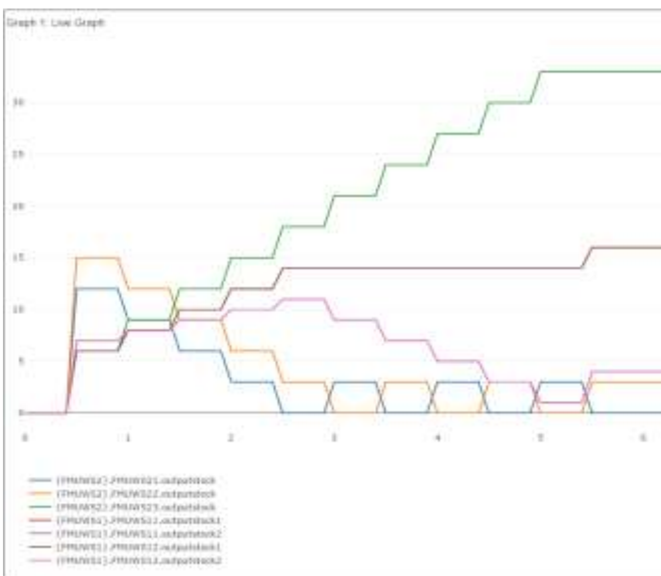


Fig. 14. Screenshot of the Output from INTO-CPS.

VI. Original Contributions and Conclusions

In this paper we used the mechanisms of category theory to specify diagrammatic models with co-simulation facilities. We introduced the concept of behavioral rule as an aggregation between a graph transformation and a behavioral action. We defined these behavioral rules by graph signatures at the metamodel level. We also implemented a simple example of diagrammatic language using the ADOxx [21] metamodeling platform. In all the phases of specification and implementation we highlighted the implementation of the constituent components of such an FMU. We performed the co-simulation of a concrete model specified with the SML language on the INTO-CPS platform.

Model transformations, if any, must be implemented in component M (Fig. 13), otherwise they could not be executed in parallel. As a result, the preconditions and postconditions should also be executed in component M. In principle, this is acceptable.

As it results from the previous observations, there are some important problems to be solved that we will deal with in future work such as: the implementation of a more complex model containing graphical transformations or the implementation of the export facility of a tool-wrapper for DSMLs implemented with ADOxx.

References

[1] M. Fowler, R. Parsons, Domain Specific Languages, 1st ed. Addison-Wesley Longman, Amsterdam, 2010.

[2] Uwe Wolter, Zinovy Diskin, The Next Hundred Diagrammatic Specification Techniques, A Gentle Introduction to Generalized Sketches, 02 September 2015 : https://www.researchgate.net/publication/253963677.

[3] D.C. Crăciunean, D. Karagiannis, Categorical Modeling Method of Intelligent WorkFlow. In: Groza A., Prasath R. (eds) Mining Intelligence and Knowledge Exploration. MIKE Lecture Notes in Computer Science, vol 11308. Springer, Cham (2018).

[4] D.C. Crăciunean, D. Karagiannis, A categorical model of process co-simulation, Journal of Advanced Computer Science and Applications(IJACSA), 10(2), (2019).

[5] C. Gomes, C. Thule, D. Broman, P.G. Larsen, H. Vangheluwe - Co-simulation: State of the art, - ACM Computing Surveys, Vol. 1, No. 1, Article 1. Publication date: January (2016).

[6] Functional Mock-up Interface for Model Exchange and Co-Simulation, Document version: 2.0.1 October 2nd 2019, https://fmi-standard.org/.

[7] INTO-CPS Tool Chain User Manual, Deliverable Number: D4.3a Version: 1.0 Date: December, 2017 Public Document, http://into-cps.au.dk.

[8] D. Karagiannis, H.C. Mayr, J. Mylopoulos, Domain-Specific Conceptual Modeling Concepts, Methods and Tools. Springer International Publishing Switzerland (2016).

[9] Dominik Bork, Dimitris Karagiannis, Benedikt Pittl, A survey of modeling language specification techniques, Information Systems 87 (2020) 101425, journal homepage: www.elsevier.com/locate/is.

[10] R. Milner, The Space and Motion of Communicating Agents, Cambridge University Press, (2009).

[11] D.C. Crăciunean, Categorical Grammars for Processes Modeling, International Journal of Advanced Computer Science and Applications(IJACSA), 10(1), (2019).

[12] Michael Barr And Charles Wells, Category Theory For Computing Science- Reprints in Theory and Applications of Categories, No. 22, 2012.

[13] Diskin Z., König H., Lawford M., 2018. Multiple Model Synchronization with Multiary Delta Lenses. In: Russo A., Schürr A. (eds) Fundamental Approaches to Software Engineering. FASE 2018. Lecture Notes in Computer Science, vol 10802. Springer, Cham.

[14] Zinovy Diskin, Tom Maibaum- Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond, ACCAT 2012.

[15] D. Plump, 'Checking graph-transformation systems for confluence', ECEASST, vol. 26, 2010. DOI: 10.14279/tuj.eceasst.26.367.

[16] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, Frank Hermann, Graph and Model Transformation General Framework and Applications, Springer-Verlag Berlin Heidelberg 2015.

[17] D. Plump, 'Computing by graph transformation: 2018/19', Department of Computer Science, University of York, UK, Lecture Slides, 2019.

[18] G. Campbell, B. Courtehoute and D. Plump, 'Linear-time graph algorithms in GP2', Department of Computer Science, University of York, UK, Submitted for publication, 2019. [Online]. Available: https://cdn.gjcampbell.co.uk/2019/Linear-Time-GP2-Preprint.pdf.

[19] FMU SDK, https://github.com/qtronic/fmusdk.

[20] Claudio Gomes, Casper Thule, Levi Lucio, Hans Vangheluwe, and Peter Gorm Larsen, Generation of Co-simulation Algorithms Subject to Simulator Contracts, https://sites.google.com/view/cosimcps19.

[21] ADOxx, https://www.adoxx.org.