A Multi-Formalism Modeling Framework: Formal Definitions, Model Composition and Solution Strategies

Hamid Mohammad Gholizadeh, Mohammad Abdollahi Azgomi *
School of Computer Engineering,
Iran University of Science and Technology, Tehran, Iran
hamid.gholizadeh@gmail.com; azgomi@iust.ac.ir

Received: 2011/02/14; Accepted: 2011/04/09 Pages: 47-62

Abstract

In this paper, we present a multi-formalism modeling framework (abbreviated by MFMF) for modeling and simulation. The proposed framework is defined based on the concepts of meta-models and uses object-orientation to overcome the complexities and to enhance the extensibility. The framework can be used as a basis for modeling by various formalisms and to support model composition in a unified manner. The structure of the framework is organized in four layers: (1) the meta-formalism layer, (2) the formalism layer, (3) the class-model layer, and (4) the model layer. The basic concepts of the framework are formally defined using object constraint language (OCL) and have been illustrated using some examples. We have also explained the model composition structure and the solution strategies of the proposed framework. A prototype tool for the proposed framework is implemented, which is briefly introduced in this paper.

Keywords: *Modeling tool, object-oriented modeling (OOM), model evaluation, multi-formalism modeling framework (MFMF)*

1. Introduction

Most of the existing modeling and simulation tools are dedicated to one or a limited number of predefined modeling or simulation languages. The witness of this claim is the information published in the Petri nets tool database available on [1]. The models constructed by these tools are not mostly interoperable with the models constructed by other tools, even if the models are based on the same modeling language. Usually, the main concern of developers is how to implement a tool for a new modeling language. As the best of our knowledge, there are few modeling tools, which are extensible for a new formal modeling language (or formalism) and new solution or simulation techniques.

On the other hand, the complexity of systems and their corresponding models is growing, so a single formalism is not enough for modeling the whole system. Therefore, we need a way to construct models that are composed of several sub models of diverse formalism types. Sometimes, parts of a model are previously constructed and can be reused to compose a new model. Obviously, this kind of model composition is rarely supported by the existing modeling tools.

The above mentioned concerns have been the motivations and aims of developing a new multi-formalism modeling framework. We have adopted meta-modeling concepts to propose a *multi-formalism modeling framework* (abbreviated by MFMF), which is flexible enough to support diverse formalisms and models in an integrated and unified modeling environment. We have used meta-modeling concepts in defining formalisms, models and model solvers. The concepts of object-orientation, such as abstraction, interfaces and encapsulation, are used to handle the complexities in the framework.

In a previous paper [2], we presented the basic ideas and preliminary results of MFMF. In this paper, we present formal definitions, model composition and solution strategies of MFMF. We will also introduce a modeling tool we have developed based on MFMF and present some examples of the implementation of formalisms in the framework.

The rest of this paper is organized as follows. In Section 2, we survey the related works. In Section 3, we present the meta-modeling structure of the proposed modeling framework, the formal definitions and model composition and solution strategies of the proposed modeling framework are given. In Section 4, some sample formalisms defined according to the framework rules are presented. In Section 5, we introduce a modeling tool we have developed to support the framework. Finally, some concluding remarks are mentioned in Section 6.

2. Related Work

There are many modeling and simulation tools, but most of them support a fixed and non-flexible model construction environment. On the other hand, some trends exist on introducing new approaches for developing multi-model multi-formalism environments. For example, Möbius [3], AToM^3 [4] and OsMoSys [5] are multi-formalism modeling tools exist in the literature. Among them, OsMoSys is closer than the others to our work in using meta-models. It is intended to support multiple formalisms in a common framework. As the best of our knowledge, this framework does not have a complete formal definition. In [4], the framework is defined in a semi-formal manner with no support for definition of new formalisms. The OsMoSys solution approach is based on a new formal language, named SPDL, which forces a modeler to learn its quite complex syntax.

The Möbius modeling tool is the result of another attempt to create a multi-formalism framework. Its idea is based on defining an abstract function interface (AFI), which is a common application programming interface (API) for adding new formalisms to the framework and using its feature [6]. Although, Möbius has interesting features for model composition and solution techniques, adding a new formalism to the framework is not an easy task. Since its first version, which supported stochastic activity networks (SANs), performance evaluation process algebra (PEPA) [7] and MoDeST [21] are the only formalisms, which have been implemented in the Möbius modeling framework.

On the other hand, AToM^3 [4], uses meta-models to support modeling by different modeling languages. The tool does not offer model solution features, so the modeler should transform the models into DEVS formalism [8] and then apply DEVS solution techniques for evaluating models. Hence, the modeler cannot use original solution techniques for models, which may be more efficient and useful.

Other well-known modeling tools (e.g. CPN Tools [9]), do not support multiple formalisms and their extensibilities are mostly limited. Some multi-formalism tools, such as SHARPE [10], support a fixed set of formalisms, solvers and simulators.

3. The Proposed Framework

We start the definition of the proposed multi-formalism framework (MFMF) by introducing a meta-modeling structure. In the framework, the model related data are organized in four layers to add flexibility and scalability in MFMF. Figure 1 depicts these four layers and their interpretation in the framework. The first top layer (i.e. Layer 0) is the most abstract layer. In this layer, models are considered as a collection of elements with some properties. We leave this layer more abstract to make the definition of diverse formalisms possible in the framework.

Layer 1, elaborates the Layer 0 definitions by specifying the elements' names, types and properties. The elements can be of types *Node*, *Edge* or *Model*. The elements of *Edge* type should be connected to other elements of the type *Node* or *Model*. An element of the type *Model* represents a sub-formalism in a formalism definition. For all elements, extra information can be annotated by properties. A formalism structure can be defined by formalism designer at this layer in the framework.

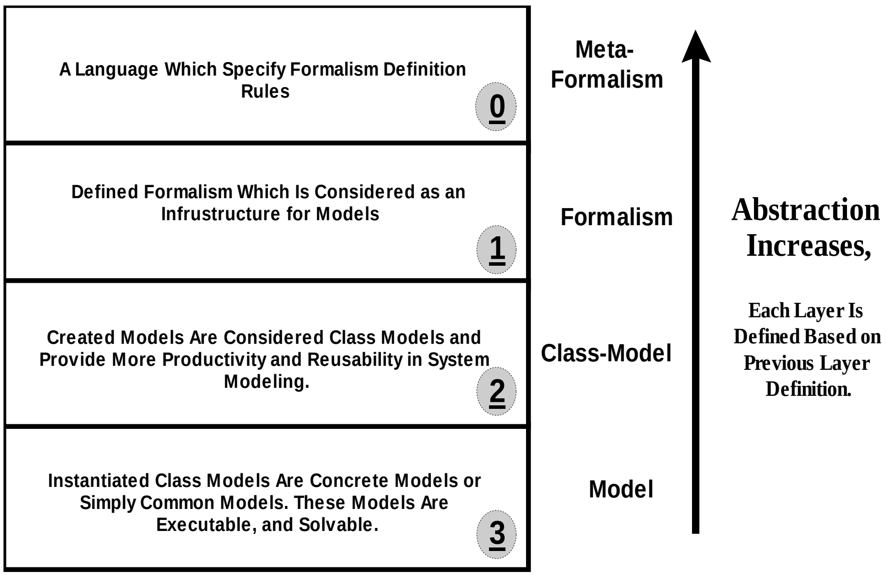

A Language Which Specify Formalism Definition Rules
0
Defined Formalism Which Is Considered as an Infrustructure for Models
1
Created Models Are Considered Class Models and Provide More Productivity and Reusability in System Modeling.
2
Instantiated Class Models Are Concrete Models or Simply Common Models. These Models Are Executable, and Solvable.
3
Meta-Formalism
Formalism
Class-Model
Model
Abstraction Increases,
Each Layer Is Defined Based on Previous Layer Definition.

Figure 1. The meta-modeling structure of MFMF

Layer 2, includes the models derived from the formalism defined in Layer 1. Models in this layer are considered as a collection of instances of the elements defined for formalism in the previous layer. For more clarification, suppose the definition of a place in Petri nets [11] and a Petri net model with five places. The place element will be defined in Layer 1 and five instances of place will be defined in Layer 2. Commonly, it seems that the models in this layer should be final solvable models. But, we have added one more layer to increase the flexibility of the framework. Models defined in Layer 2, are considered as *class-models* and they can be instantiated many times in each modeling study. Then, we need another layer as a concrete model layer, which is Layer 3. Then, the models are ready to be solved by a solution manager in the framework. In Layer 3, models are instantiated. Each class-model of Layer 2 can be instantiated several times with different parameters.

Having the above primary descriptions of the framework's meta-model structure, we continue to define these four layers' specifications formally for precise clarification of the mentioned concepts.

Formal Definitions

Now, we define the framework's meta-model structure formally. Firstly, we start to define some preliminary definitions. These definitions are necessary, since they are referred inside meta-model layers' definitions. We start by the definition of data structure.

Definition 3.1. A *data structure* in MFMF is a function $DS: PN \rightarrow T$, where *PN* is the unique name of a property (an element of the data structure) and $T \in \{PRIMARY, ENUM, P_{DS}, SET\}$ is the type of the property, where:

- $PRIMARY=\{int, float, Boolean, String\}$, is the set of primary data types.
- *ENUM*, is an enumeration type.
- P_{DS}, is a previously defined data structure.
- *SET*, is a bag (or multiset) of the elements like *W*, such that:

$$\forall w \in W, w \in \{PRIMARY, ENUM, P_{DS}\}$$

The above definition of data structure makes it possible to construct every data structure, which may exist in the definition of a formal language.

Now, we continue with the definition of elements, which is a primary component of the formalism in MFMF.

Definition 3.2. Each *element* of a formalism in MFMF is an 11-tuple: $E_F = (N_E, Img_E, T_E, F_E, P_E, C_E, A_E, ER_E, IR_E, Start_E, End_E)$, where:

- N_E, is a unique name of the element.
- Img_E, is a graphical representation of the element (provided to the related tool as a file in a standard graphics format).
- $T_E \in \{Node, Model, Edge\}$, represents the type of the element.
- F_E, is the formalism that the element belongs to.
- P_E: $PN \rightarrow T$, denotes the properties of the element, where *PN* is the name of the property and we have: $T \in \{PRIMARY, ENUM, CLASS, OBJECT, \text{FUNCTION}, \text{SET}\}$, where:
  - o *PRIMARY*, is defined as in Definition 3.1.
  - o *ENUM*, is an enumeration type.
  - o *CLASS*, is s data structure, defined as in Definition 3.1.
  - o *OBJECT*, is a reference to a data structure like *CLASS*, which is defined inside the class-model (as in the following Definition 3.5).
  - o *FUNCTION*, is the 3-tuple (I_f, O_f, C_f), where:
    - § $O_f \in K$, $I_f=2^K$, where the former defines *output type* and the latter defines the *input types*, where $K=\{PRIMARY, ENUM, CLASS, OBJECT, \text{SET}\}$.

  - § C_f, is a constraint expression for the function defined in *object constraint language* (OCL) [12].
  - § *SET*, is a bag of elements like *W*, containing all of the valid data structures defined inside F_E.
- C_E, is the OCL expression defining the element's constraints.
- A_E, is the ancestor element, which is extended by the current element, defined as follows:

$$A_E \in \bigcup \{K.E\} \qquad where \quad K \in F_E.EFR \wedge T_E = K.E.T_E$$

- ER_E is a reference to an external model, such that:

$$if\,(T_E == Model) : ER_E \in (F_E.EFR \cup F_E)$$
$$else : ER_E = \varnothing$$

- IR_E is a reference to an internal element, such that:

$$(IR_E \in F_E.E) \wedge (F_E.E \neq E_F).$$

- $START_E$, END_E are start and end nodes of the element where the element type is an arc, such that:

$$\begin{aligned} & if\,(T_E = edge): \\ & \quad (Start_E \subseteq L, End_E \subseteq L\ , \\ & \qquad L \subset \{e \mid e \in F_E.E \wedge e.T_E \neq Edge\}) \\ & \quad \wedge (Start_E \neq End_E) \\ & else : Start_E = End_E = \varnothing \end{aligned}$$

In the above definition, the element type is defined by T_E. As mentioned earlier, it can be of the type *Node*, *Edge* or *Model*. There are some properties for each element. The type of these properties may be *PRIMARY, ENUM, CLASS, OBJECT, FUNCTION* or a set of them. Class types are data structures defined inside the formalism's definition. If we define a property as an *OBJECT* type, we mean that this property refers to a class type that its definition is postponed to class-model layer. Simply, we can consider it like pointers or references in programming languages. Object type in MFMF's definition is useful in implementing some formalisms, such as coloured Petri nets (CPNs) [13] or coloured stochastic activity networks (CSANs) [14], where the modeler can define a new structure inside the model itself and then assign the type to coloured places. In some formalisms, such as SANs, CPNs and so on, there are some functions in the body of the formalism's definition. Definition of these functions as a property with the type of string is not precise. Since, we cannot define constraints. For clarity, suppose the input gate function in SANs, which can only change the marking of the nodes directly connecting to it. We need a way to express this constraint in formalism's definition in MFMF. We consider such a function as a property of the type FUNCTION and define its constraints using OCL [12]. The OCL expression in MFMF is written based on components and relationships depicted in Figure 2. After defining an element in MFMF, we can present the formal definition of formalism, which is a collection of the elements of MFMF.

Definition 3.3. A *formalism* (or *formal modeling language*) *F* in MFMF is a 6-tuple *F*={*N*, *Img*, *P*, *E*, *EFR*, *ADS*}, where:

- *N*, is a unique name for the formalism in MFMF context.
- *Img* is a graphical representative of the formalism (provided to the related tool as a file in a standard graphical format).
- *P*, is the properties of the formalism (defined as in Definition 3.1).
- *E*, is a set of the formalism's elements, where $E \neq \emptyset$, $E=\{E_1, E_2, \ldots E_n\}$, where E_i is defined as in Definition 3.2.
- *EFR*, are references to other defined formalisms in the context, which we may want to use their elements as an ancestor in the current formalism definition ($EFR \in 2^{F_{UMF}}$).
- *ADS*, is a collection of data structures defined inside the formalism according to Definition 3.1.

Till now, we have defined the formalism in MFMF. Considering the above definition, we can summarize a formalism definition in MFMF in a unified modeling language (UML) like class diagram as shown in Figure 2. In this figure, it is clearly illustrated that the formalism can contain other formalisms, too. A formalism in MFMF is a collection of elements with some properties for each. Therefore, class-models are some instantiated elements from these elements including some elements of the node or model types, which are connected to each other by some elements of edge types.

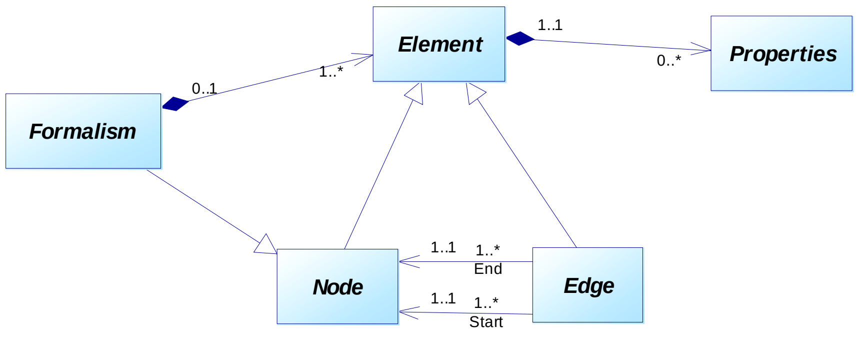


Figure 2. The meta-model of formalism definition in MFMF as a class diagram

Now, we present the definition of class-model. For this purpose, first we need to present some preliminaries. We define each element of the class-model formally and then provide a formal definition for the class-model itself.

Definition 3.4. An element of a class-model, E_{MC}, in MFMF is a 5-tuple E_{MC}=(*M*, *T*, *P*, *VI*, *A*), where:

- *M*, is the class-model that this element belongs to.
- *T*, is the type of the element. It is one of the permitted types defined in the corresponding formalism of class-model of the element ($T \in M.F.E$).

- $P=\{p_1, p_2, \ldots, p_n\}$, is the properties of the element. There is a bijective function[1] $f{:}P \rightarrow D_{T.P}$ ($D_{T.P}$ is the domain of P_E as in Definition 3.2), which enforce a one-to-one relationship between P and $D_{T.P}$, and P_i is a 4-tuple (N_i, VA_i, VI_i, A_i), where:
  - $N_i = f(p_i).PN$, is the name defined for the property in the formalism definition.
  - VA_i, is the value of the property that should be compatible with the type of the property defined in the formalism definition ($TYPE(VA_i) = f(p_i).T$).
  - $VI_i \in \{private, public\}$, is the property's visibility in the class-model.
  - $A_i = \{ReadOnly, WriteOnly\}$, is accessibility modifier.
- VI and A, define the visibility and the accessibility of the element, respectively, and their definitions are same as VI_i and A_i.

Now, we continue the MFMF's definitions by defining the class-model formally:

Definition 3.5. A *class-model*, M, is defined as a 5-tuple $M=(N, F, E, P, O)$, where:

- N, is a name for the class-model.
- F, is a reference to the corresponding formalism that the class-model is based on.
- E, is the set of elements, each one defined as in Definition 3.4.
- P, is the set of class-model properties $P=\{p_1, p_2, \ldots, p_n\}$, and there is a bijective function like $g{:}P \rightarrow D_{T.P}$ ($D_{T.P}$ is the domain of P as in Definition 3.3) which apply a one-to-one relationship between P and $D_{T.P}$, and p_i is a 4-tuple (N_i, VA_i, VI_i, A_i), where:
  - N_i, is the name of the property($N_i = g(p_i).PN$)
  - VA_i, is the value of the property. It should be compatible with the type of the property defined in the formalism definition ($TYPE(VA_i) = g(p_i).T$)
  - VI_i, A_i, are defined as in Definition 3.4.
- O, is a data structure defined as in Definition 3.1.

According to the above definitions, it is obvious that the class-model itself may have some properties same as its elements. It means the extra information can easily be annotated to the elements of the class-model or to the class-model independently. In Section 4, we will present some sample formalism definitions in MFMF to demonstrate these definitions' applicability in defining different formalisms in the framework.

Model Composition and Solution Strategies

It is possible to define various composed models in the framework. We will define its possibility in the formalism definition of MFMF. For example, we can define hierarchical stochastic activity networks (HSANs) [16] or Petri nets formalism composed of CPNs or SANs as submodels. A submodel in a composed model is connected by an arc to at least one container node or to another submodel. Semantics of this connection is defined by means of a *relation function* written in a high-level programming language (e.g. Java). This function defines how the values in a submodel can be affected by the values in the container model and vice versa. Defining communication by using a programming language between the container model and its submodels makes the relationship more flexible. The relation function is executed by the solvers of MFMF. We define a relation function formally as follows.

1. simultaneously injective and surjective

Definition 3.6. A *relation function* for an edge, *E*, which connects *A* (a submodel element) to *B* (another submodel element), is a function like R_E: $V_1 \rightarrow V_2$, such that:

$$V1 = \{p.VA_i \mid p \in (A.P \cup (e.p \mid e \in A.E)) \wedge p.A_i = ReadWrite \wedge p.VI_i = public\} \cup$$
$$\{p.VA_i \mid p \in (B.P \cup (e.p \mid e \in B.E)) \wedge p.A_i = ReadWrite \wedge p.VI_i = public\}$$
$$TYPE(V2) = TYPE(V1)$$

Definition 3.7. A *relation function* for an edge, *E*, which connects *A* (a simple (non-submodel) element) to *B* (a submodel element) is a function like R_E: $V_1 \rightarrow V_2$, such that:

$$V1 = \{p.VA_i \mid p \in A.P \wedge p.A_i = ReadWrite\} \cup$$
$$\{p.VA_i \mid p \in (B.P \cup (e.p \mid e \in B.E)) \wedge p.A_i = ReadWrite \wedge p.VI_i = public\}$$
$$TYPE(V2) = TYPE(V1)$$

According to the above definitions, the relation function can change the values of the properties of the connected elements. If two submodels connected by an edge with the relation function, their *ReadWrite* public properties can be affected by that function during the solution phase. If one end is a simple element, then its properties can be affected as well.

The execution and manipulating the property values is the task of the solution manager. In the previous section, we defined the strategy of MFMF for defining structure of a formalism in the framework, though we did not mention how MFMF handles the behavior or the execution rules of the formalism. The solution manager alongside some solvers is responsible for the formalism's execution rules inside the framework. They execute models and generate desired results out of the solution procedure, according to the formalism's execution rules and the relation functions. The solution manager is aware of the relation functions and the solvers know about the formalism execution rules. As well as most collaborative systems, a manager (i.e. the solution manager) orchestrates all solver components to solve the mode. The solution manager synchronizes the solvers and manages the data communication between the container model and its submodels.

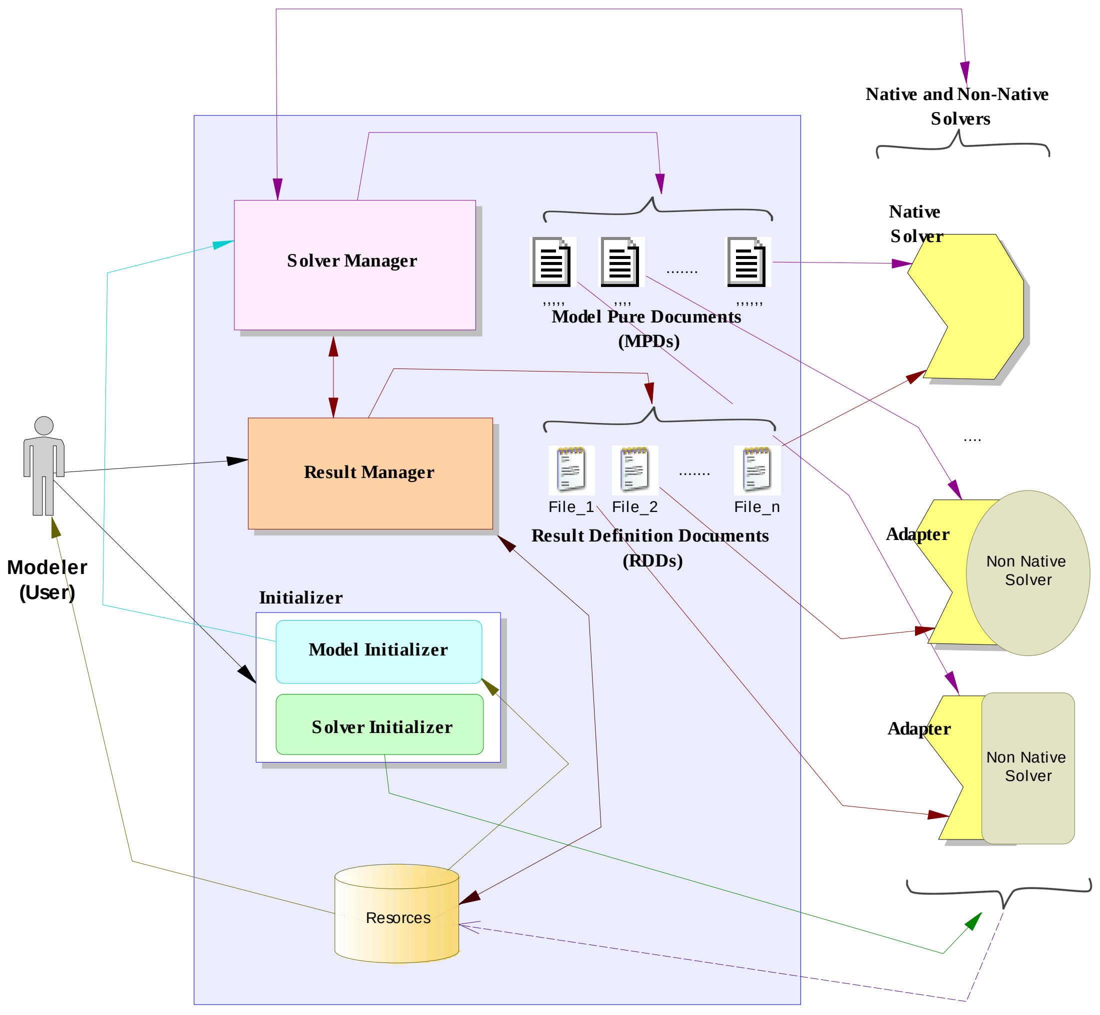


Figure 3. Framework of the solver system

Figure 3 depicts a general view of the architecture of the solution system. The system includes *Native* and *Non-Native Solvers*, *Adapters*, *Initializer*, *Result Manager*, *Solution Manager* and the framework's resources. We comment briefly on each of these parts and discuss their roles in the solution process.

The solvers may be implemented exclusively for the framework or external solvers may be employed. We call the former series, *native solvers* and the latter series, *non-native solvers*. For adapting non-native solvers into the framework, we use some adapters. These adapters force the external solvers to support some important functionalities like taking the care of the relation functions, execution, stopping and resuming of the model execution during the solution phase.

Initializer includes *Solver Initializer* and *Model Initializer*. The former should initialize and prepare the necessary *Native* or *Non-Native Solvers* for collaborating in the solution process. *Model Initializer* initializes and prepares the models and submodels' parameters to make a concrete model ready for execution. The values for the parameters may be provided by the user.

Result Manager asks for the desired results from the user by providing a suitable user interface. The requested results of the solution can be separated into two parts. They may be directly related to some values of the model or submodels' properties, or they may be generated by executing predefined functions. Before execution process, *Result Manager* generates the *Result Definition Documents* (RDDs) to deliver to the solvers. These documents tell the solvers the values of the interested properties.

Solution manager gets the initialized model files and separates them into *Model Pure Documents* (MPDs). MPDs are documents containing the submodel's information. For each submodel one MPD is generated in addition to one sole MPD for the container model. These MPDs include the information of only one submodel and will be delivered to the solvers. Their contents are suitably annotated to completely meet the solution process integration requirement as done collaboratively by the solution manager and other solvers.

4. Sample Formalisms Described in the Framework

In this section, we show how we can formally describe Petri nets, generalized stochastic Petri nets (GSPNs) [17] and hierarchical stochastic activity networks (HSANs) as sample atomic and composed formalisms in MFMF. We describe these formalisms based on the formal definitions presented in Section 3.

a. Descriptions of Petri Nets and GSPNs

In this section, we describe Petri nets in MFMF according to its formal definition [11]. Petri nets are consisting of three basic elements including place, transition and arc. Each instance of these elements may have a caption. A place may contain some tokens. The caption and the tokens can be shown, by a string and an integer, respectively, as the properties of the element. A Petri net model can be described in MFMF as the following:

Definition 4.1. **A** Petri net in MFMF is described as follows:

$$F_{Petrinet} = \{N, Img, P, E, EFR, ADS\}$$
$$N = "petrinet"$$
$$Img = "img\ /\ petrinet.svg"$$
$$E = \{E_P, E_T, E_{Arc}\}$$
$$P = EFR = ADS = \varnothing$$

where each element is described as below:

$$E_P = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}$$
$$N = "place", \quad Img = "img\ /\ place.svg", \quad F = F_{petrinet}$$
$$T = Node, \quad P = \{("token", "int"), ("caption", "String")\}$$
$$A = C = ER = IR = Start = End = \varnothing$$

$$E_T = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}$$
$$N = "transition", \quad Img = "img\ /\ transition.svg",$$
$$T = Node, \quad P = \{("caption", "String")\}$$
$$F = F_{petrinet}, \quad A = C = ER = IR = Start = End = \varnothing$$

$$\begin{aligned}
&E_{Arc} = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}\\
&\quad N = "arc", \quad Img = "img\ /\ Arc.svg", \quad F = F_{petrinet}\\
&\quad T = Edge, \quad P = \{("caption", "String")\}\\
&\quad Start \in \{E_p, E_T\} \quad End \in \{E_p, E_T\}\\
&\quad C = \text{context } E_{Arc} \text{ inv:}\\
&\qquad (\text{self}.Start.N = place \text{ implies self}.End.N = transotion) \text{ and}\\
&\qquad (\text{self}.Start.N = transition \text{ implies self}.End.N = place)\\
&\quad A = ER = IR = \varnothing
\end{aligned}$$

The OCL expression defined for the arc element in Definition 4.1 implies that each arc can connect a place to a transition and vice versa. Then, a place (respectively, a transition) cannot be connected to another place (respectively, a transition) directly. The formalism in MFMF can be defined by extending the previously defined formalism in the context. Then, according to this concept, we will describe GSPNs. This description is based to the formal definition of GSPNs [17].

Definition 4.2. We describe GSPNs in MFMF as below:

$$\begin{aligned}
&F_{GSPN} = \{N, Img, P, E, EFR, O, C\}\\
&\quad N = "GSPN" \quad Img = "img\ /\ GSPN.svg"\\
&\quad P = C = O = \varnothing \quad E = \{E_P, E_{TT}, E_{IT}, E_{Arc}\}\\
&\quad EFR = \{F_{petrinet}\}
\end{aligned}$$

$$\begin{aligned}
&E_P = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}\\
&\quad N = "place", \quad Img = "img\ /\ place.svg",\\
&\quad T = Node, \quad A = F_{petrinet}.E_P \quad F = F_{GSPN}\\
&\quad P = ER = IR = C = Start = End = \varnothing
\end{aligned}$$

$$\begin{aligned}
&E_{TT} = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}\\
&\quad N = "timed\_transition", \quad F = F_{GSPN}\\
&\quad Img = "img\ /\ timed\_transition.svg",\\
&\quad T = Node, \quad A = F_{petrinet}.E_T,\\
&\quad P = \{("rate", Function(\varnothing, "float", C_{rate})\}\\
&\qquad C_{rate} = \text{context } E_{TT} \text{ post: self}.F{-}{>}\ forAll(E = E@\text{pre})\\
&\quad ER = IR = C = Start = End = \varnothing
\end{aligned}$$

$$\begin{aligned}
&E_{IT} = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}\\
&\quad N = "itransition", \quad Img = "img\ /\ itransition.svg",\\
&\quad F = F_{GSPN}, \quad T = Node, \quad A = F_{petrinet}.E_T,\\
&\quad P = \{("selProb", "float")\}, \quad ER = C = IR = Start = End = \varnothing
\end{aligned}$$

$$\begin{aligned}
&E_{Arc} = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}\\
&\quad N = "Arc", \quad Img = "img\ /\ Arc.svg", \quad F = F_{GSPN}\\
&\quad T = Edge, \quad A = F_{petrinet}.E_{Arc}\\
&\quad Start \in \{E_p, E_{TT}, E_{IT}\} \quad End \in \{E_p, E_{TT}, E_{IT}\}
\end{aligned}$$

C = context E_{Arc} inv:
let $st : E$ =self.$Start$, $en : E$ =self.End in
($st.N = place$ implies ($en.N = itransition$ or
$en.N = timed\_transition$)) and
(($en.N = itransition$ or $en.N = timed\_transition$)
implies $en.N = place$)
$P = ER = IR = \varnothing$

According to the formal definition of GSPNs, there are two types of transitions: *timed* and *immediate*, which are respectively named *timed_transition* and *itransition* in the above descriptions. These elements alongside the place element inherit their descriptions from the elements of Petri nets as in Definition 4.1. The rate of each timed transition is defined as a property of the type FUNCTION. The post-condition for this function guaranties no change on any elements of the model. The definition of arc in GSPNs is defined same as Petri nets and can possibly connect the permitted elements to each other.

b. Descriptions of HSANs

HSANs are an extension of SANs formalism supporting hierarchical models in their definition [16]. Here, we demonstrate how we can describe HSANs as a sample of composed formalism in MFMF. We omit the descriptions of some elements for brevity.

Definition 4.3. HSANs in MFMF is defined as follows:

$$F_{HSAN} = \{N, Img, P, E, EFR, O, C\}$$
$$N = "HSAN", \quad Img = "img / HSAN.svg", \quad O = \varnothing$$
$$P = ("data", OBJECT), \ EFR = \{F_{GSPN}, F_{petrinet}\}$$
$$E = \{E_P, E_{TA}, E_{IA}, E_{IG}, E_{OG}, E_{SubSAN}, E_{Arc}, E_{SubSANArc}\}$$

For brevity, we only define $E_{IG}, E_{SubSAN}, E_{SubSANArc}$. The descriptions of other elements are straightforward.

$E_{IG} = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}$
$N = "IGate"$, $Img = "img / IGate.svg"$, $F = F_{SAN}$
$T = "Node"$,
$P = \{("guard", FUNCTION(\varnothing, boolean, C_{predicate}))$,
$("fun", FUNCTION(\varnothing, \varnothing, C_{fun}))$, $("caption", String)\}$
$C_{predicate}$ = context E_{IG} post: self.$F->$ $forAll(E = E$@pre)
C_{fun} = context E_{IG} post:
let
$adjacentE$:set(E) = self.$F.E->$ select(e:E | $E.End$ = self),
$allToken$:set(int)= self.$F.E->$ $collect(P.token)$
in
$allToken ->$ exclude($adjacentE->$ $collect(P.token)$) =
($allToken ->$ exclude($adjacentE->$ $collect(P.token)$))@pre
$ER = C = A = IR = Start = End = \varnothing$

$$E_{SubSAN} = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}$$

$$N = "subSAN", \quad Img = "img / subSAN.svg",$$

$$F = F_{HSAN}, T = Model, \; A = F_{petrinet}.E_P, \; ER = F_{HSAN}$$

$$P = \{(data, OBJECT)\} \quad C = IR = Start = End = \varnothing$$

$$E_{SubSANArc} = \{N, Img, T, F, P, A, ER, IR, C, Start, End\}$$

$$N = "SubSANArc", \; Img = "img / SubSANArc.svg",$$

$$T = "Edge", \quad A = F_{petrinet}.E_{Arc}, \quad F = F_{HSAN}$$

$$P = \{("RelFun", Function(\varnothing, \varnothing, C_{RelFun}))\}$$

$$Start \in \{E_p, E_{SubSAN}\} \;\; End \in \{E_p, E_{SubSAN}\}$$

$$C_{RelFun} = Default$$

$C =$ context $E_{SubSANArc}$ inv:

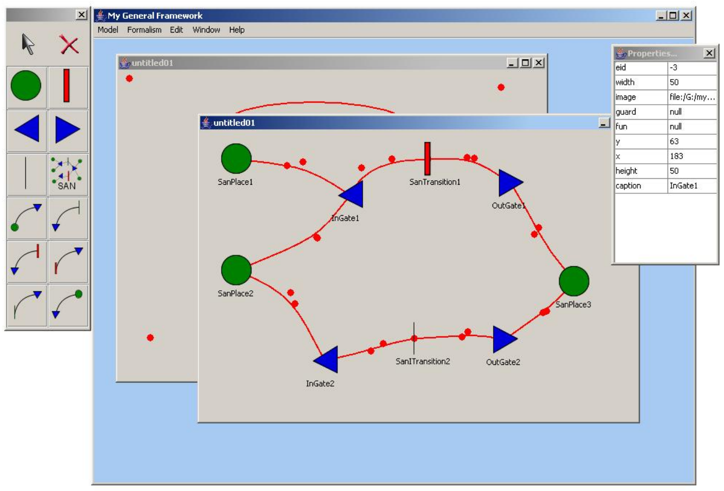
```
        let
          $st : E$=self.$Start$, $en : E$=self.$End$
        in
        ($st.N = place$ implies $en.N = subSAN$) and
        ($st.N = subSAN$ implies $en.N = place$)
```

$$ER = IR = \varnothing$$

In the above definition, $E_{SubSANArc}$ has a property named *RefFun*. As we discussed previously, this is a special property that defines how a submodel is connected to a container model in a composed model. In the formal definition of HSANs, the relationship of submodels with the container model is simply defined by fusion places [16], but by the relation function strategy, there is more options than just simple mapping between fusion places. Here, the constraint for the relation function is defined as default. It means that the default constraint is previously defined in the relation function definition. Then, we avoid redundancy in defining it in all relation functions unless we want to specify extra constraints for that. *IG* represents *Input Gate* and has two property of the type FUNCTION, consequently defining guard and gate functions of the element. The guard function cannot change the state of the model and the gate function can only change the token properties of the places directly connected to the IG element. These constraints for the function are defined in OCL expressions, too.

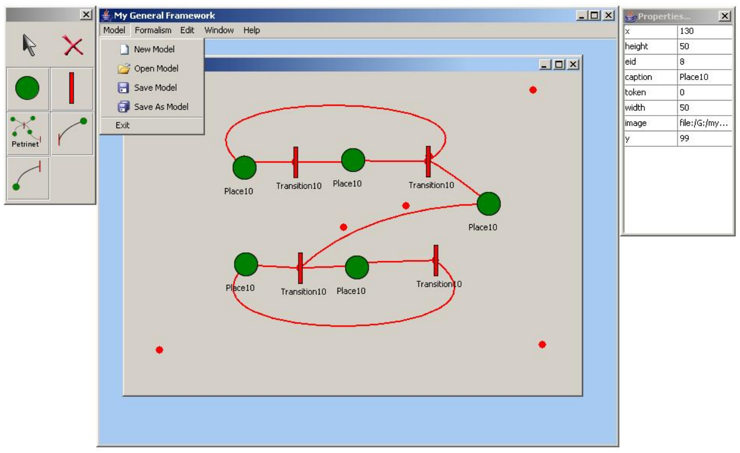
5. A Tool for the Framework

A modeling framework needs a supporting tool. We have implemented a tool for MFMF to support its features. Some snapshots of the tool are shown in Figure 4 through 6. Extensible markup language (XML) is an infrastructure for storing all kinds of data, including formalism definition, class-model definition, transient data between solvers and solution manager and so on in our framework. Using XML simplifies importing the models that are not defined exclusively for the framework using the MFMF tool.



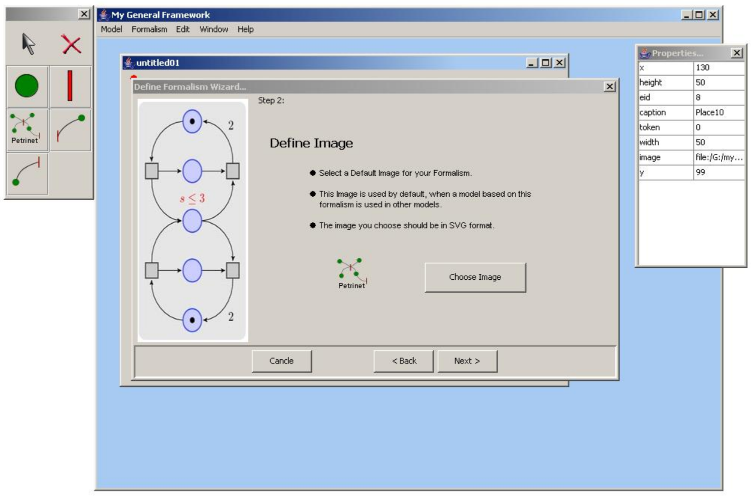
eid	-3
width	50
image	file:/G:/my...
guard	null
fun	null
y	63
x	183
height	50
caption	InGate1

Figure 4. A sample SAN model in the GUI of the MFMF tool



x	130
height	50
eid	8
caption	Place10
token	0
width	50
image	file:/G:/my...
y	99

Figure 5. A producer/consumer model in the MFMF tool



Step 2:

Define Image

- Select a Default Image for your Formalism.
- This Image is used by default, when a model based on this formalism is used in other models.
- The image you choose should be in SVG format.

x	130
height	50
eid	8
caption	Place10
token	0
width	50
image	file:/G:/my...
y	99

Figure 6. The formalism definition wizard of the MFMF tool

By implementing a simple extensible stylesheet language transformation (XSLT) document, the models compatible with Petri net markup language (PNML) [18] can easily be adapted by the framework. However, we have used scalable vector graphics (SVG), which are XML-based images for representing the images inside the MFMF tool.

We have used Java programming language and JavaEE [19] features in the implementation of the MFMF tool. Its design is based on the *model-view-controller* (MVC) model [20], since it should be an interactive and event-driven application. We have exploited Java Reflection API to manipulate meta-data layers dynamically. Then, after defining new formalism, which is provided by using a wizard, it generates necessary XML files, compiles and deploys necessary Java class files and makes new defined formalism accessible inside the framework without a need to recompilation of the tool or even restarting the tool. By using Java native interface (JNI), we have implemented adapters that can communicate external implemented solvers in other languages. Other features of JavaEE let us to distribute solution process over computer networks, too. The tool is so customizable that one can easily switch between different defined formalism for making a composed model.

6. Conclusions

In this paper, we introduced a meta-modeling approach in defining a new multi-formalism modeling framework (MFMF). Using four abstract layers for organizing the framework's data structure and using object-oriented techniques in its definition makes the framework flexible, scalable and interoperable. Therefore, the framework is extensible by a large number of formalisms. The innovative approach in defining a framework for diverse formalisms provides an infrastructure for defining a unified modeling environment for constructing atomic models or homogeneous or heterogeneous composed models. A new method named relation function is proposed for the composed models to handle communications between models and submodels. Object constraint language (OCL) expressions are used in the framework's structure to precisely define formalisms. We illustrated the applicability of the framework by defining some sample formalisms using features of the framework. We discussed the techniques for implementing a tool for the framework, too.

In future, we intend to continue the work by completing the implementation of the tool for the framework. Also, we will introduce a method for formal description of the solution techniques in the framework.

7. References

[1] "Petri Nets Tool Database," URL: http://www.informatik.uni-hamburg.de/ TGI/ PetriNets/ tools/db.html, Visited: 2011-08-28.

[2] H. Mohammad Gholizadeh and M. Abdollahi Azgomi, "A Meta-Model Based Approach for Definition of a Multi-Formalism Modeling Framework," International Journal of Computer Theory and Engineering (IJCTE), Vol. 2, No. 1, 2010, pp. 87-95.

[3] "The Möbius Tool", URL: https://www.mobius.illinois.edu/, Visited: Visited: 2011-08-28

[4] "AToM^3: A Tool for Multi-Formalism Meta-Modelling," URL: http://atom3.cs.mcgill.ca/, Visited: Visited: 2011-08-28.

[5] V. Vittorini, M. Iacono, N. Mazzocca and G. Franceschinis, "The OsMoSys Approach to Multi-formalism Modeling of Systems," Software and Systems Modeling (SoSyM), Vol. 3, No. 1, Springer, 2004, pp. 68-81.

[6] "Möbuis Manual," PERFORM Group, University of Illinois at Urbana-Champaign, 2007.

[7] G. Clark and W. H. Sanders, "Implementing a Stochastic Process Algebra within the Möbius Modeling Framework," Proc. of the First Joint PAPM-PROBMIV Workshop, Springer-Verlag, 2001, pp. 200-215.

[8] S. Palaniappan, A. Sawheny and H. S. Sarjoughian, "Application of DEVS Framework in Construction Simulation," Proc. of the 2006 Winter Simulation Conference, Institution of Electrical and Electronics Engineers, Piscataway, N.J., 2006, pp. 2077-2086.

[9] "CPN Tools Homepage," URL: http://cpntools.org/, Visited: 2011-08-15.

[10] K. S. Trivedi, "The SHARPE Tool and the Interface (GUI)," URL: http://people.ee.duke.edu/~chirel/IRISA/sharpeGui.html, visited: 2011-08-20.

[11] J. L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall, 1981.

[12] "OCL 2.2 Specification," The Object Management Group, 2010.

[13] K. Jensen, Coloured Petri Nets: A High Level Language for System Design and Analysis, Lecture Notes in Computer Science, Vol. 483, Advances in Petri Nets 1990, Springer-Verlag, 1991, pp. 342-416.

[14] M. Abdollahi Azgomi and A. Movaghar, "Towards an Object-Oriented Extension for Stochastic Activity Networks," Proc. of the 10th Workshop on Algorithms and Tools for Petri Nets (AWPN'03), Eichstaett, Germany, Sept. 26-27, 2003, pp. 144-155.

[15] A. Movaghar, "Stochastic Activity Networks: A New Definition and Some Properties," Scientia Iranica, Vol. 8, No. 4, 2001, pp. 303-311.

[16] M. Abdollahi Azgomi and A. Movaghar, "Modeling and Evaluation with Object Stochastic Activity Networks," Proc. of the 1st Int. Conf. on Quantitative Evaluation of Systems (QEST'04), Enschede, The Netherlands, Sept. 27-30, IEEE CS Press, 2004, pp. 326-327.

[17] M. Ajmone Marsan, G. Balbo and G. Conte, "A Class of Generalized Stochastic Petri Nets for Performance Evaluation of Multiprocessors Systems" ACM Transactions on Computer Systems, Vol. 2, No. 2, 1984, pp. 93-122.

[18] "Systems and Software Engineering - High-level Petri Nets," Part 2: Transfer format, ISO/IEC 15909-2, 2011.

[19] "Java EE at a Glance," URL: http://www.oracle.com/technetwork/java/javaee/overview/ index.html, Visted: 2011-09-20

[20] S. Burbeck, Applications Programming in Smalltalk-80 (TM): How to Use Model-View-Controller, 1992.

[21] H. Bohnenkamp, H. Hermanns, I. P. Katoen and R. Klaren, "The MoDeST Modeling Tool and its implementation," Proc. of the Computer Performance Evaluation Modelling Techniques and Tools (TOOLS'03), Lecture Notes in Computer Science, Vol. 2794, Springer-Verlag, 2003, pp. 116-133.