Interfaces and Binding in Component Based Development of Formal Models

G. Franceschinis
Università del Piemonte Orientale, Italy
giuliana@unipmn.it

M. Gribaudo
Università di Torino, Italy
marcog@di.unito.it

M. Iacono
Seconda Università di Napoli, Italy
mauro.iacono@unina2.it

S. Marrone
Seconda Università di Napoli, Italy
stefano.marrone@unina2.it

F. Moscato
Seconda Università di Napoli, Italy
francesco.moscato@unina2.it

V. Vittorini
Università di Napoli "Federico II", Italy
vittorin@unina.it

ABSTRACT

Component based modeling is of great importance for building and analyzing models of real systems. It is based on a well known paradigm which makes use of abstraction and composition. In this paper we focus on abstraction, by describing a practical approach to the definition of very simple interface models allowing the substitution of components within composed multiformalism models. The work extends the OsMoSys methodology and relies on meta-modeling. This paper does not discuss formal aspects about interface theory and components interaction, but focuses on the problem of building component models in practice with the ultimate goal of solving them by using (the existing) analysis tools. The paper formally extends the OsMoSys conceptual model in order to introduce model interfaces and to provide some rules for interface compatibility. The paper also describes some steps towards the full definition of mechanisms for interface binding and their implementation.

Categories and Subject Descriptors

I.6 [**Computing Methodologies**]: Simulation and Modeling—*Simulation Support Systems*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms

Design, Languages, Performance, Reliability

Keywords

Formal Models, Interface, Binding, Metamodeling, Dependability, Performability

1. INTRODUCTION

The vision of component software engineering is to provide reusable, off-the-shelf software components for designing large applications from existing building blocks [22], hence providing a general solution for the reuse problem. The basic idea is that a software component is a "unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [19]. The aim of the research work (partially) described in this paper is to realize this vision into the development of formal models. More, we want to introduce the possibility of dynamically changing or choosing parts of the model during the solution steps (e.g. a similar concept is implemented by dynamic Web Service substitution and orchestration [12]).

This paper does not discuss formal aspects about interface theory and components interaction, but focuses on the problem of building component models in practice with the ultimate goal of solving them by using (existing) analysis tools. In order to achieve these goals we extend the OsMoSys Modeling Methodology with a notion of model interface, interface compatibility and binding, and then we introduce some of the mechanisms to implement them. Indeed, a powerful technique in component programming is being able to implement multiple interfaces on an object. By this technique multiple classes may implement the same interface, and a single class may implement one or more interfaces. By implementing interfaces, a component can provide functionalities to any other component requiring that interface. This allows for the interchangeability of different versions of a component without affecting the whole system. In this paper we define the concept of interface for models considered as components. Our goal, besides promoting model reuse, is to globally improve the OsMoSys modeling approach for the analysis of complex systems by allowing on-the-fly (re)definition of parts of a complex model, i.e. while solving the model.

The scientific community has studied this topic from different points of view leading to different research directions. One research direction concerns the application of formal methods and notations to object oriented and component based approaches [4]. Other approaches, as e.g. [7], formalize the difference between interface models and behavior models by means of a definition of an "Interface alge-

VALUETOOLS 2009 October 20-22, 2009, Pisa, Italy

Table 1: The OsMoSys modeling stack

Level	OsMoSYs layer	Description
M3	Metaformalisms	Languages to define formalisms.
M2	Model Metaclasses	Formalisms to build models
M1	Model Classes	Model specifications
M0	Model Objects	Model instances

bra" putting the stress on stateless interfaces. Objects and component typing problem is addressed in [5] where a λ-calculus has been specialized in order to deal with types and in [1] that focuses on object oriented typing theory (with another extension of λ-calculus, too). The research direction of Model Driven Engineering (MDE) [15] studies models, languages and model transformations. It is aimed at formalizing component based theory of models, in order to improve reusability. Several works aim at this objective also by means of a model typing theory [21, 16].

The approach proposed in this paper focuses on the problem of practically building component models: in particular we addressed the problem of building up a coherent methodology and framework for multiformalims-based modeling and analysis.

The paper is organized as follows. Section 2 summarizes the main concepts on which the OsMoSys methodology is based and the notations used. Section 3 extends the methodology by introducing the definitions of Pure Interface and Interface Implementation. Section 4 introduces some of the rules that must be applied to guarantee compatibility between component models and interfaces, and the concept of early and late binding in OsMoSys. Section 5 discusses the implementation of these mechanisms within the OsMoSys framework. Finally Section 6 contains closing remarks and some hints about future work.

2. FORMALISMS AND METAMODELING

The goal of the OsMoSys (Object-baSed multi-formalism MOdeling of SYStems) research project is the definition of a methodology and the implementation of a software framework for the development and the analysis of multi-formalism models. The OsMoSys Modeling Methodology (OMM) has been formerly introduced in [11, 23] and it is supported by the DrawNET tool [13, 14] for the development of models, and by the OsMoSys Multisolution Framework (OMF) [17] for the analysis of complex multi-formalism models; moreover such approach has been effectively used to model and analyse complex critical systems [9, 10].

OMM defines a conceptual framework based on object orientation concepts and metamodeling, in which formalisms are classes and new formalisms can be easily introduced or defined by inheritance from existing ones. Table 1 shows the four layers of meta-modeling [3, 8] on which the OsMoSys approach is based.

As introduced in section 1, Model Driven Engineering defines a (meta)-modeling language stack [3] and heavily relies on model transformations [6]. This approach historically focuses its attention on Model Driven Software Development [20] as an extension of OMG's Model Driven Architecture [18]. A growing interest on building UML profiles must be cited: some of these profiles have become OMG standard as MARTE [2], a UML profile for modeling reliable and mission-critical systems. Although there are few Model Driven research works that deal with formal methods and languages (e.g. [24] describes formal languages as a particular cases for source and target languages in model transformation), at the best of our knowledge these works do not focus on the definition of a Model Driven Engineering oriented framework for multiformalism modeling and multisolution analysis.

Fig.1 depicts the OsMoSys conceptual model which represents the main concepts involved in the modeling methodology. It is organized into a set of packages mainly including:

- Level M3: *Metafomalisms*. This package provides the two languages used to define formalisms. The Formalism Description Language (FDL) is used to define modeling languages. FDL allows to define formalism elements type (*ElementType*), element properties (*PropertyType*) and constraints (*ConstraintsType*) used to define grammar rules. The Result Definition Language (RDL) is used to define the properties (e.g. performance indices) that may be associated to a modeling formalism and to its elements. Hence the properties refer to element types (*ElementTypeRef*) and to element properties (*PropertyTypeRef*).

- Level M2: *Formalisms* and *Results*. These packages include the languages (built by M3 meta-languages) used to describe the modeling formalisms of interest (*Formalisms*), and the languages used to describe and ask for indices evaluation (*Results*). Results Types are introduced by the DataType package. A special role among formalisms is played by the Bridge Formalisms used to build composed multi-formalism models. The concept of Bridge Formalism is introduced by the *MultiFormalism* package. *Formalism*s are composed by *Element*s. In particular the Elements of a Bridge Formalism may be Formalisms or composition *Operator*s. Operators define the composition semantics and In, Out and In-Out *Parameters* that models can exchange. The *inner class* relation among Formalism and Element limits the scope of the Elements inside the Formalisms they belong to. Hierarchies of Formalisms and hierarchies of Elements may be built in order to introduce new Formalisms also from the existing ones.

- Level M1: *Classes*. This package introduces the *Model Class*es. A *Model Class* describes a family of models compliant with a specified Formalism and sharing a common structure (built by using the elements introduced by the Formalism). A Model Class defines a parametric model, in the sense that the values of a non-empty subset of its properties are not specified. Hence the model needs to be instantiated in order to be solved.

- Level M0: *Objects*. Instantiated models are called *Model Object*s and basically they are Model Classes where all properties are defined.

A hierarchy of formalisms is briefly exemplified in Fig.2.

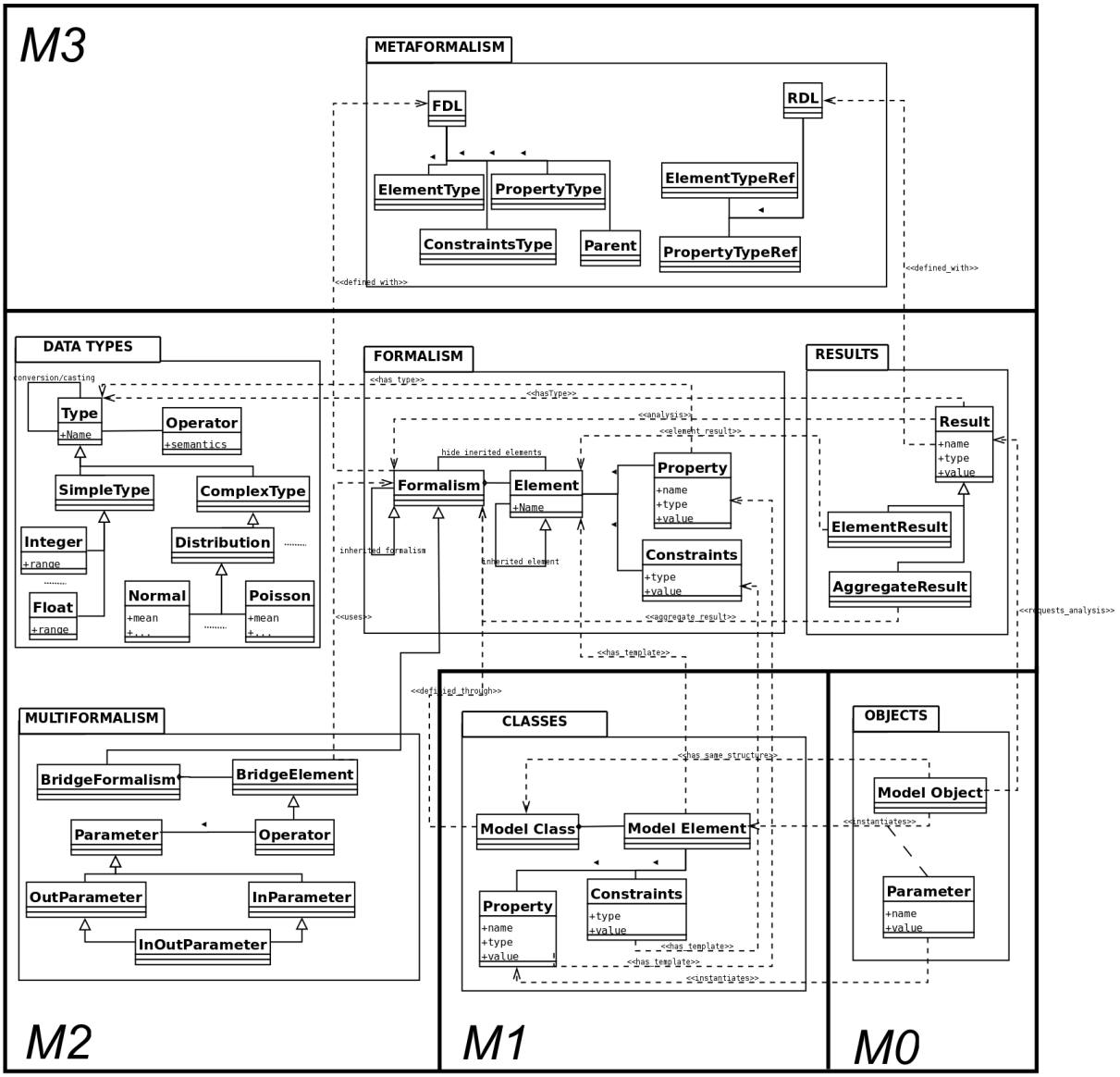

Figure 1: OsMoSys Conceptual Model

As explained before, Formalisms are Metaclasses (Level M2) and they include modeling formalisms as Petri Nets, Fault Tree etc., but also other formalisms useful to define them, for example by inheritance. This is the case of the *GraphBased* formalism in Fig.2. *GraphBased* is defined by using FDL and it is the base class for all the formalisms which define Elements like nodes and edges. Hence, *Petri Nets* inherits from *GraphBased*, the Elements *Place* and *Transition* inherit from the Element *Node* of *GraphBased* , and the Element *Arc* inherits from *Edge*. *Place* adds the Property *token*. *GSPN* in turn is derived by *Petri Nets* and the Elements *Immediate Transition* and *Timed Transition* inherit from *Transition*. *Timed Transition* adds the *rate* property; *Immediate Transitions* instead add the priority and weight properties.

An example of Results specification (defined over the *Petri Net* and *GSPN* formalisms) is reported in Fig.3 where the *Liveness* Aggregate Result and the *Throughput* Element Result are shown.

The following notations will be used throughout the paper. The set of element types included in a given formalism $\mathcal{F}$ is denoted $\mathcal{E}_{\mathcal{F}}$; an alternative notation $(\mathcal{F}, \mathcal{E})$ may be used to couple a formalism with the set of its element types. Each element type $e \in \mathcal{E}_{\mathcal{F}}$ has a name and one or more "properties" that are used to specify the attributes of the related elements. Let F be a formal language, and let $(\mathcal{F}, \mathcal{E})$ be the corresponding Metaclass with its non-empty set of element types. A Metaclass in OsMosys, as metaformalism in traditional approaches of Language Engineering, can be inherited from another one: inheriting a Metaclass implies the possibility to add or to hide formalism element or elements properties. The formal definition of Metaclass inheriting mechanisms are out of the scope of this paper. Formally a Model Class $MC_{\mathcal{F}}$ is a triple (N, S, SM) where:

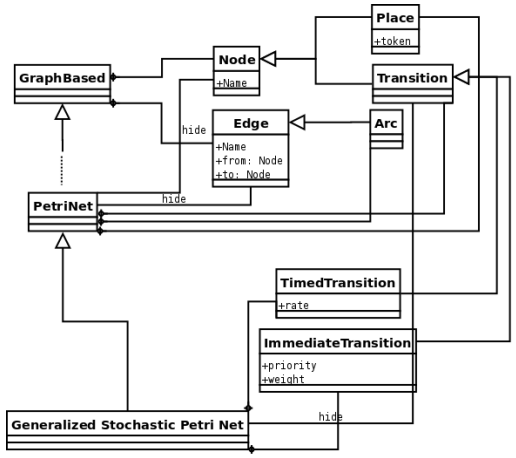

Figure 2: An example of Formalisms Hierarchy

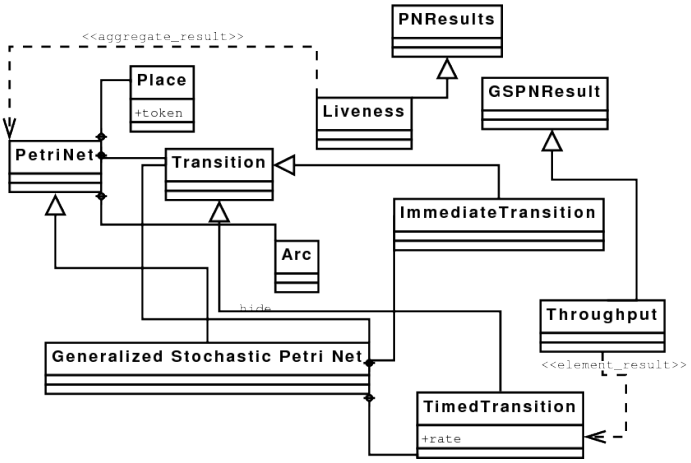

Figure 3: An Example of Results specification

- N is the class name i.e., the type name of all the instances of the class;
- S is the class structure i.e. set of Elements (and their Properties and Constraints) that is common to all the models of type N.
- SM is the set of the (sub)models that a model contains.

The structure S of the class is a set of element instances of element type $\mathcal{E} \cup SM$ that are simply called "elements" instead of "element instances", abusing notation. For each element $e \in S$, $Type(e)$ is the element type of e (hence $Type(e) \in (\mathcal{E} \cup SM)$). For example, if a Model Class is created using the Petri Nets formalism, its structure will comprise instances of element types from $\mathcal{E}_{\mathcal{PN}}$ (i.e. instances of $Place$, $Transition$ and Arc): if $P1$ is a place instance, then $Type(P1) = Place$.

S is the union of two different sets:

$$S = External_S \cup Internal_S;\ External_S \cap Internal_S = \oslash$$

- $External_S$ is the *interface* of the class i.e., the subset of the elements, possibly belonging to some (sub)models, that have the role of connectors or ports to the external environment;
- $Internal_S$ is the subset of elements that are encapsulated by the class.

Both external and internal elements in S may have a set of attributes denoted P_S. $P_S = EP_S \cup IP_S$, where EP_S is the set of the attributes that may be set when a model of type N is instantiated (called "parameters" in the following), and IP_S is the set of attributes that are statically defined by the class definition.

The distinction between $External$ and $Internal$ elements does not include the concepts of Model Class Interface and Interface Binding. A more suitable definition of Model Class Interface will be given in Section 3. Further details on the modeling methodology can be found in [23].

3. COMPONENTS AND INTERFACES

In this Section we provide an extension to OMM in order to introduce a new definition of interface of a Model Class.

We define here the mechanisms by which multiple Model Classes may implement the same interface and a single Model Class may implement more than one interface. The main drawback of the Model Class interface definition reported in Section 2 is that $External_S$ is a set of elements, hence it strictly couples the interface of a Model Class to the particular formalism and structure through which the model is expressed. For example, let us suppose that $SimpleProcessor_{\mathcal{F}}$ is a Model Class representing a CPU and that its interface consists of one element:

- through which it exchanges information with the external environment, or
- that can be used for model manipulation and transformation purposes (e.g. Petri Nets transitions superposition).

Let us suppose that $SimpleProcessor_{\mathcal{F}}$ is used to build a complex composed model. When a different version of $SimpleProcessor_{\mathcal{F}}$ is needed and available, it should be possible to substitute this sub-model without modifying the whole model. The new version of $SimpleProcessor_{\mathcal{F}}$ could be implemented by using a different formalism $\mathcal{F}$ (e.g. Queuing Networks instead of Petri Nets) or it could have a different structure S even if it is expressed by the same formalism (e.g. two different Petri Nets PN1 and PN2: PN2 is a version of PN1 that takes into account the presence of a cache). This substitution is not possible if the interface of the Model Class changes when varying the implementation of the Model Class itself. This difficulty is overcome by introducing interface elements, as informally exemplified below.

In Fig. 4 (a) and (b) two simple Generalized Stochastic Petri Nets (GSPN) models are shown. They represent a processor read operation without and with caching features respectively.

The average time required by a read operation is represented by the inverse of transition *T1* firing rate in the GSPN model of the processor without caching features (Fig. 4 (a)). In the GSPN model of the processor with caching features (Fig. 4 (b)), a datum (a cache line) is stored in the cache with probability $Prob\_Cache$. The access to a cache line is

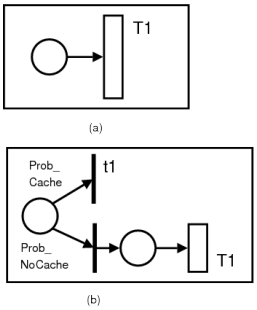

Figure 4: Two simple GSPN models

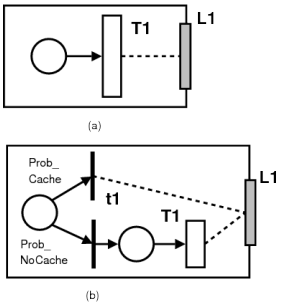

Figure 5: An example of interface element

assumed to be a not time consuming operation (modeled by the immediate transition *t1*), otherwise the read operation takes the same (average) time assumed in the first model.

According to the interface definition provided in Section 2 the transition *T1* in Fig. 4 (a), may be specified to be an interface element of this Model Class. In Fig. 5 (a) a new element has been added to the Model Class and the transition *T1* is associated to this particular element called *interface element* (through a dotted-line connecting them). This new element is labeled *L1* and it is explicitly addressed when the Model Class interface is involved in composition and/or substitution instead of the transition *T1*.

Similarly, in Fig. 4 (b), the transitions *t1* and *T1* may be specified to be interface elements: they both represent the occurrence of a read operation and they are mutually exclusive. Hence, in Fig. 5 (b) they are associated to the same interface element *L1*.

In order to formally define interfaces and interface elements, the concept of *Pure Interface formalism* and *Pure Interface Model Class* are introduced.

The Pure Interface formalism $\mathcal{IF}$ is defined at level M2 of the OsMoSys stack. $\mathcal{IF}$ inherits from *GraphBased* and its set of element types is:

$$\mathcal{E}_{\mathcal{IF}} = \{IE\}$$

where IE is derived by the element type *node* of *GraphBased*. Each interface element must have a unique identifier, that is one of its properties. We shall denote such identifier $IE.name$: in Fig. 5 this corresponds to the label *L1*.

Definition 1: Pure Interface Model Class ***(PI).***

A *Pure Interface Model Class* (or *Pure Interface* for short) is a Model Class $MC_{\mathcal{IF}}$ compliant with the Pure Interface formalism $\mathcal{IF}$.

A Pure Interface is thus defined at level M1. The set S of a Pure Interface only contains elements of type IE. Similarly to abstract classes in object oriented programming a Pure Interface cannot be instantiated to produce a Model Object: it must be "implemented" by the structure of a "concrete" Model Class. In other words a Pure Interface can be substituted by a Model Class which "implements" the interface.

To this aim, the structure of a Model Class must contain interface elements and proper arcs, as in the example of Fig. 5. This is obtained introducing at level M2 the formalism *Interfaceable* ($\mathcal{INF}$), which in turns inherits from $\mathcal{IF}$ adding an element type derived by the element type *edge* of *GraphBased*. Hence the set of element types in $\mathcal{INF}$ is:

$$\mathcal{E}_{\mathcal{INF}} = \{IE, IFEdge\}$$

A modeling formalism suitable to develop component based models should inherit from $\mathcal{INF}$. Hence the PN formalism used to develop the models in Fig. 5 is derived from $\mathcal{INF}$.

$\mathcal{INF}$ defines two element types: interface element (IE) and interface edge ($IFEdge$). $IFEdge$ is a (not oriented) arc used to connect interface elements and elements of formalisms inheriting from $\mathcal{INF}$.

The relationships among the introduced formalisms and elements are shown in Fig.6. To state if a Model Class may implement a Pure Interface the property *BindingConstraint* of the element IE is used. This will be discussed in the next section.

Now, let $\mathcal{D}$ be a Formalism which inherits from $\mathcal{INF}$ and $MC_{\mathcal{D}}$ a Model Class compliant with $\mathcal{D}$. Hence, the set of elements in the Model Class may contain IE and $IFEdge$ elements, in addition to the elements defined in the formalism $\mathcal{D}$. If S is the set of elements in the Model Class newly defined in the formalism $\mathcal{D}$, $S_{\mathcal{D}} = \{s \in S : Type(s) \in \mathcal{E}_{\mathcal{D}} - \mathcal{E}_{\mathcal{INF}}\}$, and $S_{\mathcal{INF}}$ the set of elements in the Model Class defined in the formalism $\mathcal{INF}$, $S_{\mathcal{INF}} = \{s \in S : Type(s) \in \mathcal{E}_{\mathcal{INF}}\}$, then $S = S_{\mathcal{D}} \cup S_{\mathcal{INF}}$, where $S_{\mathcal{D}} \cap S_{\mathcal{INF}} = \oslash$. We denote:

- $E_{S_{\mathcal{D}}}$ the (sub)set of elements in $S_{\mathcal{D}}$ that are intended to be interfaced with the external models;
- $EN_{S_{\mathcal{INF}}}$ the (sub)set of IE elements in $S_{\mathcal{INF}}$, i.e. $EN_{S_{\mathcal{INF}}} = \{e \in S_{\mathcal{INF}} | Type(e) = IE\}$
- $EA_{S_{\mathcal{INF}}}$ the (sub)set of $IFEdge$ elements in $S_{\mathcal{INF}}$, i.e. $EA_{S_{\mathcal{INF}}} = \{e \in S_{\mathcal{INF}} | Type(e) = IFEdge\}$

We can give now a definition of a *Well Structured Model Class*, which is a Model Class that can be bound to a Pure Interface.

Definition 2: Well Structured Model Class ***(WSMC).***

$MC_{\mathcal{D}}$ is *Well Structured* iff it is a Model Class compliant with the Metaclass $\mathcal{D}$ inheriting from $\mathcal{INF}$ and:

1. $EA_{S_{\mathcal{INF}}} \subseteq (E_{S_D} \times EN_{S_{\mathcal{INF}}})$
2. $\forall e \in E_{S_D}, \exists\, ie \in EN_{S_{\mathcal{INF}}} \colon a = (e, ie) \in EA_{S_{\mathcal{INF}}}$
3. $\forall ie \in EN_{S_{\mathcal{INF}}}, \exists\, e \in E_{S_D} \colon a = (e, ie) \in EA_{S_{\mathcal{INF}}}$

The first condition says that the set $EA_{S_{\mathcal{INF}}}$ is a set of arcs between the elements in E_{S_D} and the $IFEdge$ elements

in $EN_{S_{\mathcal{INF}}}$. The second condition says that for each element e in E_{S_D} at least one element ie exits in $EN_{S_{\mathcal{INF}}}$ so that e and ie are connected by the arc $a = (e, ie) \in EA_{S_{\mathcal{INF}}}$. Finally the third condition says that for each interface element ie at least one element e exists which is connected to it by an arc in $EA_{S_{\mathcal{INF}}}$.

In particular, each element $e \in E_{S_D}$ may be connected to more than one interface element $ie \in EN_{S_{\mathcal{INF}}}$ and viceversa.

Notice that being Well Structured does not mean for a Model Class that it is able to implement a Pure Interface. The notion of Well Structuredness simply implies that in the structure of the Model Class all interface elements correspond to the interface elements of the Pure Interface and that each interface element has to be connected at least to one element belonging to the model structure. The "Implements" relation between Model Classes and Pure Interfaces is introduced in the next Section.

4. CONSTRAINTS AND BINDING

In order to define the "Implements" relationship, it should be possible to state if the interface of a Model Class is compatible with a given Pure Interface. The definition of IE in the Pure Interface Formalism in Section 3 includes the property *BindingConstraint* (Fig.6). It is used to introduce a kind of interface typing which allows for binding mechanisms definition and implementation.

The constraints defined on interface elements may belong to the following three categories:

1. Constraints on formalisms element types (*ElementTypeCon*);
2. Constraints on the data types of the elements properties (*PropertyTypeCon*);
3. Constraints on the data types of the results elements (*ResultTypeCon*).

Let be $(\mathcal{F}_1, \mathcal{E}_1), \cdots, (\mathcal{F}_n, \mathcal{E}_n)$ n formalisms and let $e_{ij} \in \mathcal{E}_i, j = 1, \cdots, p$ an element of $\mathcal{E}_i$. Let $e_{ij}.p_k, k = 1, \cdots, q$ and $e_{ij}.r_l, l = 1, \cdots, r$ be respectively a property and a result of e_{ij}.

Let el_{MC} be an element of a WSMC, and let us indicate with $el_{MC}.p_e$ and with $el_{MC}.r_f$ respectively one of its results an properties.

The three kinds of constraints are expressed by propositional logic. If $\mathcal{P}$ is a logic proposition built by means of logical connectives, the above constraints have the following form:

1. $\mathcal{P}(Type(el_{MC}) = e_{ij})$;
2. $\mathcal{P}(Type(el_{MC}.p_e) = X)$;
3. $\mathcal{P}(Type(el_{MC}.r_f) = X)$;

where X is a name of a Type in the Package *Types* in Fig.1. Examples of the three constraints are the following:

1. $ElementTypeCon = PN.Place$ OR $QN.Service$
2. $ElementPropertyCon = float$ OR int
3. $ElementResultCon = int$

Let ie_{MC} and ie_{PI} be interface elements of a Well Structured Model Class MC and of a Pure Interface PI.

Inside MC, ie_{MC} is connected to Model Class Elements by means of $IFedge$s. Let be cel_{MC} one of these elements.

We can say that constraints defined on ie_{PI} are *verified by* one or more ie_{MC} iff *all* cel_{MC} elements verify the constraints above mentioned. We indicate this with:

$ie_{MC} \models constr(ie_{PI})$.

Notice that, thanks to formalisms (elements) inheritance, the relation $Type(el_{MC})$ also allows the verification of the constraint $ElementTypeCon = PN.Place$ if $Type(el_{MC}) = GSPN.Place$.

The same is for properties and results types if proper casting functions exist which translate, for example, Normal distribution to float (for example: the *meanvalue*).

Constraints as they are defined, are only related to elements in Pure Interfaces Model Classes. If a Well Formed Model Class exists such that:

1. the number of its ie_{MC} is equal to the number of the ie_{PI};
2. all constraints defined by ie_{PI}s are satisfied by ie_{MC}s

this model class is said to *implement* the Pure Interface Model Class.

Formally,

Definition 3: Implementation of a Pure interface.

Let $IE_{PI} = \bigcup_i (ie_{PI_i})$ and $IE_{MC} = \bigcup_i (ie_{MC_i})$ be the sets of IEs respectively in PI and in MC. We say that

$$MC \text{ implements } PI \text{ iff:}$$

$\exists I : IE_{MC} \rightarrow IE_{PI}$ such that:

1. I is bijective;
2. $\forall ie_{MC} \in IE_{MC}, ie_{MC} \models constr(I(ie_{MC}))$.

Hence an implementation of a Pure Interface must "match" all its interface elements and the correspondence between an interface element in the pure interface and interface elements in the implementation must be 1 : 1 (since the function I is bijective); moreover, all constraints of each $ie_{PI} = I(ie_{MC})$ have to be verified by (all) the elements connected to the corresponding interface element ie_{MC} in MC.

For simplicity, we use labels for mapping elements between Pure Interfaces and Well Formed Model Classes: IEs in a Pure Interface will be mapped to IEs in a Well Structured Model Class with the same label. Formally speaking, in the current implementation we require that:

$$\forall ie_{MC} \in IE_{MC}, ie_{MC}.name = I(ie_{MC}).name \ .$$

When allowing the derivation of new Interface Elements from the basic ones, some additional constraints on function $I()$ could be required, for example that the corresponding Interface Elements according to $I()$, must be of the same (more specific) type, or that the value of a given attribute of such elements must be equal (e.g. this could be useful in case the value of such attribute expresses the type of the data exchanged through that interface).

Interface Model Classes **cannot** *be instantiated* (i.e. no Model Object can be created from them). They are interfaces and they must eventually be implemented by a Model

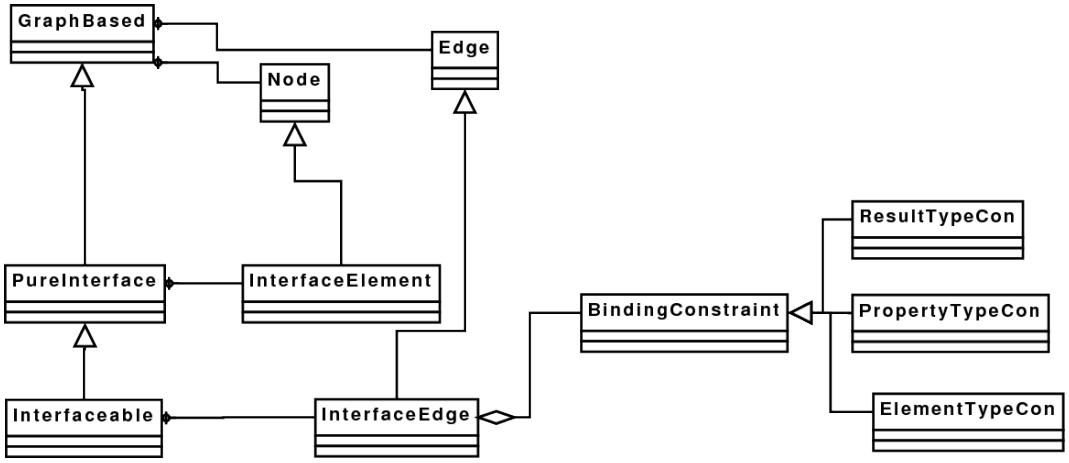

Figure 6: The Interface Hierarchy

Class. Fig. 7 shows an example of *Implements* function, consistent with the above definitions: the Model Classes in Fig. 7(b), (c) and (d) are three alternative implementations of the Pure Interface in Fig. 7(a); the correspondence between Interface Elements is represented by means of matching labels in the figure.

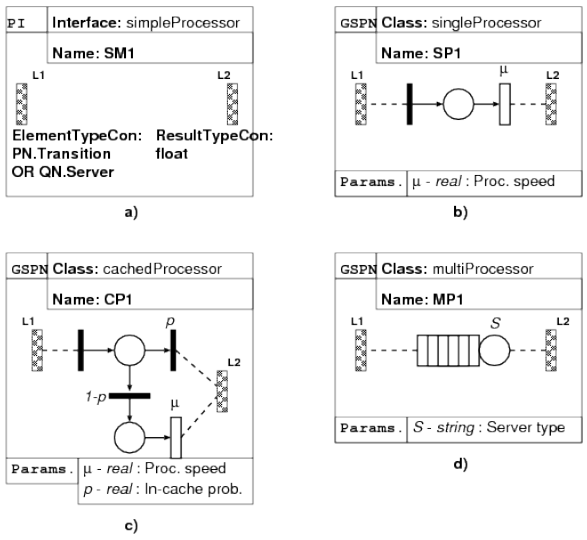

Figure 7: The *Implements* relationship

The *singleProcessor* Model Class is compliant with a GSPN Metaclass inheriting from $\mathcal{IF}$, and it implements a single processor model of a simple machine. It has a parameter represented by the rate μ (Fig. 7(b)). The *cachedProcessor* Model Class is compliant with a GSPN Metaclass inheriting from $\mathcal{IF}$ and it implements a model of a processor in which data can be in a cache (and then can be read immediately with probability **p**) or not (and can be read in a time $1/\mu$ with probability **1-p**). Thus, it has two parameters: μ and **p** (Fig. 7(c)). The *multiProcessor* Model Class is instead a queue model of a multi processor system, thus it is compliant with a different Metaclass inheriting from $\mathcal{IF}$, e.g. a QN Metaclass. Its only parameter is **S** which is a composite parameter, including all the information (service rate, number of servers, scheduling policy) on the server used in the queue (Fig. 7(d)). We recall that the parameters must be instantiated when the Model Objects are created from the Model Classes. It is easy to verify that the Model Classes *singleProcessor*, *cachedProcessor* and *multiProcessor* are Well Structured and that they all implement the *SimpleProcessor* Interface Model Class.

Notice that in Fig.7(c), two elements are linked to the interface element L2. When multiple $IFEdge$s are connected to an IE, its constraints must be verified for *all* the linked Model Class Elements.

Interface Model Classes can be part of a larger model and represent placeholders for alternative implementations: in Fig. 8 an example is shown representing a parallel machine consisting of two processors on which it is possible to distribute the workload arriving from the environment. The model contains four sub-models: the models of the processors (in the middle) are Pure Interfaces, whereas the arrival of new processing requests and the corresponding replies are modeled by means of two simple GSPN Model Classes. The two pure interfaces can be implemented by one of the model classes in the bottom part of the figure. Examples of criteria for the substitution are given in the next section.

5. IMPLEMENTATION

In order to implement the interface binding, the OsMoSys Core level of the OMF [17] has been integrated with new dedicated software components. A general description of how the OMF operates is illustrated in Fig. 9, where the new components are shadowed.

Models in OsMoSys are solved by generating a solution process, that is a workflow (expressed in SPDL, Solution Process Description Language) in which all actions needed to perform the solution of a complex multiformalism model are described. The solution process is generated by the SPDL Compiler which analyses the model and a user query describing what results have to be produced. The OsMoSys Core is composed by a WorkFlow Engine (WFE) that orchestrates solvers and other OsMoSys components to enact solution processes, a Result Manager (RM) that stores and supplies all partial results obtained during the enactment of a solution process, and an Instancer that performs the instantiation of Model Classes to obtain the Model Objects.

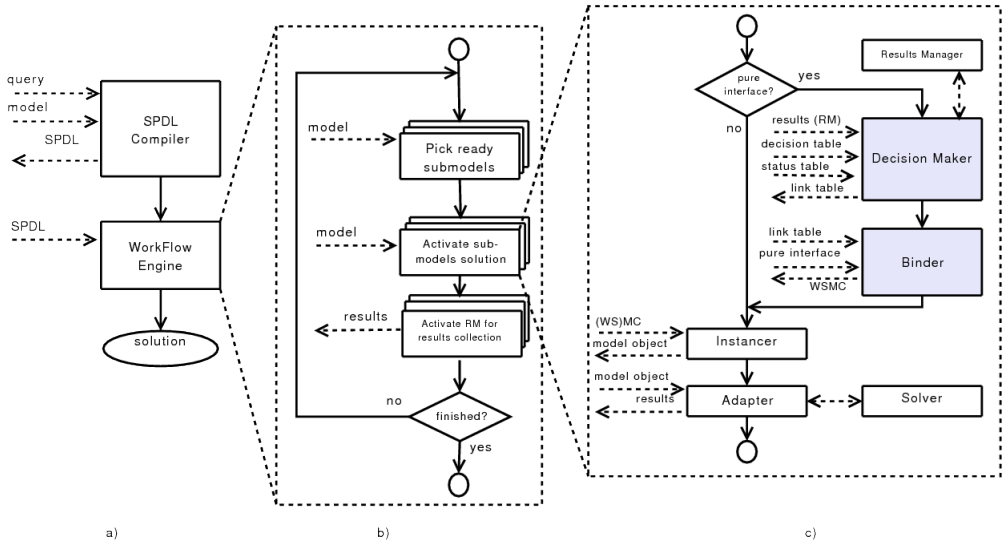

Figure 9: Operations in OsMoSys

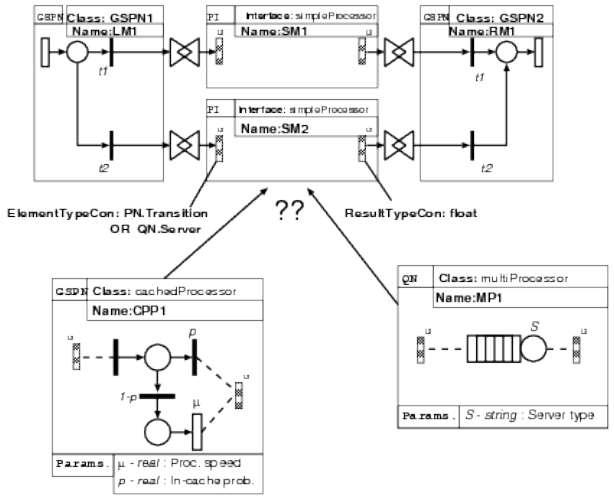

Figure 8: An example of Binding

In order to provide interface binding the architecture of the OMF Core has been extended by adding the Decision Maker and the Binder components (Fig. 9 c)): the Decision Maker is in charge of deciding which Model Class may be used to implement a Pure Interface, while the Binder checks if the binding constraints specified by a Pure Interface are verified by the Model Class indicated by the Decision Maker and, if all constraints are satisfied, it performs the substitution.

When a solution process is generated by the SPDL Compiler, it is ready to be executed by the WFE (see Fig. 9 a)). The WFE identifies submodels that are ready to be solved (Fig. 9 b)) and executes the proper activities specified in the related fragments of SPDL. At the end of each fragment executions the WFE activates the RM that collects and stores partial results.

For each submodel the related SPDL fragment specifies the activities that need to be executed to solve it: the simplest case of fragment is graphically described in Fig. 9 c). If the submodel is a Model Class it is instantiated by using the parameters values specified by the user and solved. If the submodel is a Pure Interface a binding is needed to replace the Pure Interface by a Well Structured Model Class which implements it.

To this aim three tables have been introduced.

- Status Table (ST);
- Decision Table (DT);
- Link Table (LT).

ST describes the status of: a) the system on which the solution process is being executed (e.g. the WFE workload, the availability of computing nodes etc.) and b) the solution process (e.g. the values of some relevant data of the workflow). These information are generated by the WFE and some of them may be used by the user when compiling the Decision Table.

DT contains the user's indication about the Model Classes that should be involved in the binding phase and the conditions that will drive the choice among the specified alternatives. Hence, this information must be provided by the user when submitting to the OMF a model which contains Pure Interfaces.

An example of DT (related to the model in Fig. 8) is reported in Fig. 10. This DT consists of two parts that say which are the candidates to implement the two Pure Interfaces (SM1 and SM2) in the middle of Fig. 8.

As for SM1 (see the first tag DIRECTIVE) the user wants that it is replaced by the Model Class CP1 if the value of the throughput of the transition $t1$ (tag NUMERICCONDITION) evaluated by solving the model LM1 is greater than 10 OR by the Model Class MP1 if this value is in the interval $[7, 10]$. The Model Class is compliant with the GSPN formalism, whereas the Model Class MP1 is compliant with the QN formalism. Notice that a default Model Class is also specified (SP1, compliant with the GSPN formalism). It will be used, for example, if the throughput of $t1$ is less

```
<DECISIONTABLE>
  <DIRECTIVE INSTANCEABLE="SM1"
  DEFAULTSUBMODELNAME="SP1"
  DEFAULTSUBMODELTYPE="GSPN">
    <NUMERICCONDITION VARIABLE="LM1.t1.throughput"
    KIND="RESULT">
      <GT VALUE="10" SUBMODELNAME="CP1"
      SUBMODELTYPE="GSPN"/>
      <BTW INFVALUE="7" SUPVALUE="10" SUBMODELNAME=
      "MP1" SUBMODELTYPE="QN"/>
    </NUMERICCONDITION>
  </DIRECTIVE>
  <DIRECTIVE INSTANCEABLE="SM2"
  DEFAULTSUBMODELNAME="SP2"
  DEFAULTSUBMODELTYPE="GSPN">
    <BOOLEANCONDITIONTRUE VARIABLE="StatusVar1"
    KIND="STATUS" SUBMODELNAME="MP1"
    SUBMODELTYPE="GSPN"/>
  </DIRECTIVE>
</DECISIONTABLE>
```

Figure 10: An example of Decision Table

than 7. In this example the Model Class used to replace the Pure Interface SM1 will be established by the Decision Maker after the model LM1 is solved, according to the value of a result obtained from the Result Manager.

As for SM2 (see the second tag DIRECTIVE) the user wants that it is replaced by the Model Class MP1 according to the value of a status variable, a default is also specified (SP2). In this case the decision depends on a value that will be contained into the ST table.

LT is the output of the Decision Maker, it says to the Binder which are the Model Classes that should implement the Pure Interfaces. Hence, LT contains an entry for each Pure Interface of the model. Each entry specifies the name and the type of the Model Class that implements that interface.

Notice that we are supposing that the candidates Model Classes are Well structured. This condition is verified when building the Model Class.

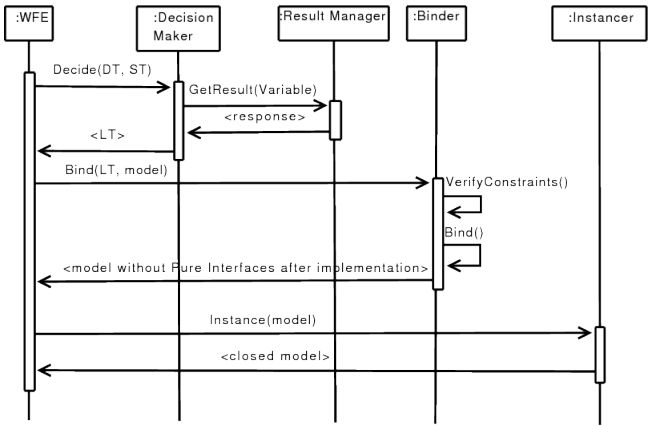

Figure 11: Binding Sequence Diagram

The UML Sequence Diagram in Fig. 11 details the interactions between OsMoSys components that implement the binding. According to the DT, for each entry the Decision Maker either examines ST to evaluate the current value of status variables or requires the Result Manager in order to evaluate the current value of result variables. The Decision Maker produces the LT, that is used by the Binder. The Binder examines the model, searching for a PI: whenever the search is successful, its name is used to access the LT and check the correctness of the instantiation. The candidate implementation interface constraints are checked against the PI Constraints and, in case of success, the interface instantiation is performed. If all interface instantiations are successful, the Binder produces a parametric model, which is then passed to the Instancer.

6. CONCLUSIONS

In this paper we focused on the problem of supporting component based definition of multi-formalism models. The possibility of allowing the re-use of components expressed in different specification languages, posed new challenges that were tackled with the definition of model interfaces. In particular, we introduced the concept of constraints, to restrict the composition only to meaningful and formally identifiable cases.

Many points still have to be clarified, and need further investigation. Nevertheless, the methodology proposed in this paper serves as a basis for further studies in this direction. The adoption of model interfaces allows the modeler to create models with components that can possibly be specified at run-time, creating more flexible specifications that can dynamically adapt to match tradeoffs between quality of the results and solution efficiency.

Future works are firstly aimed at completing the proposed approach, both under methodological and software framework aspects. In particular further development of interface typing and constraints expressions are planned.

Future works will also include the application of the proposed methodology to real-world problems. For what concerns OMF, support for both interface typing and runtime components selection will be enhanced to allow for more complex expressions in constraints and decision table specifications.

Acknowledgements

This work has been partially supported by grant MIUR-PRIN 2007J4SKYP.

7. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] S. Bernardi, J. Merseguer, and D. C. Petriu. Adding dependability analysis capabilities to the marte profile. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 736–750. Springer-Verlag, 2008.

[3] J. Bezivin. In search of a basic principle for model driven engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V(2):21–24, 2004.

[4] M. Büchi and E. Sekerinski. Formal methods for component software: The refinement calculus

perspective. In *Proceedings of the Second Workshop on Component-Oriented Programming (WCOP), volume 5 of TUCS General Publication, pages 2332, Short version in ECOOP97 workshop reader LNCS 1357*, pages 23–32. Springer-Verlag, 1997.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[6] K. Czarnecki and S. Helsen. Classification of model transformation approaches, 2003.

[7] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 148–165, London, UK, 2001. Springer-Verlag.

[8] J.-M. Favre. Foundations of meta-pyramids: Languages vs. metamodels – episode ii: Story of thotus the baboon1. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[9] F. Flammini, M. Iacono, S. Marrone, N. Mazzocca, and V. Vittorini. A multiformalism modular approach to ertms/etcs failure modelling. *submitted to: IEEE Transactions on Dependable and Secure Computing*, 2009.

[10] F. Flammini, S. Marrone, N. Mazzocca, and V. Vittorini. A new modeling approach to the safety evaluation of n-modular redundant computer systems in presence of imperfect maintenance. *to be published in: Reliability Engineering and System Safety*, 2009.

[11] F. Franceschinis, M. Gribaudo, M. Iacono, N. Mazzocca, and V. Vittorini. Towards an object based multi-formalism multi-solution modeling approach. In *Proc. of the Second International Workshop on Modelling of Objects, Components, and Agents (MOCA'02), Aarhus, Denmark, August 26-27, 2002 / Daniel Moldt (Ed.)*, pages 47–66. Technical Report DAIMI PB-561, Aug. 2002.

[12] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras. Dynamic service substitution in service-oriented architectures. In *Proc. IEEE Congress on Services - Part I*, pages 101–104, 2008.

[13] G. Gribaudo, M. Iacono, M. Mazzocca, and Vittorini. The osmosys/drawnet xe! languages system: A novel infrastructure for multi-formalism object-oriented modelling. In *ESS 2003: 15th European Simulation Symposium And Exhibition*, 2003.

[14] M. Gribaudo, D. C. Raiteri, and G. Franceschinis. Drawnet, a customizable multi-formalism, multi-solution tool for the quantitative evaluation of systems. In *QEST*, pages 257–258, 2005.

[15] S. Kent. Model driven engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 286–298. Springer-Verlag, 2002.

[16] T. Kühne. Matters of (meta-) modeling. *Software and Systems Modeling (SoSyM)*, 5(4):369–385, December 2006.

[17] F. Moscato, F. Flammini, G. D. Lorenzo, V. Vittorini, S. Marrone, and M. Iacono. The software architecture of the osmosys multisolution framework. In *ValueTools '07: Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, pages 1–10, 2007.

[18] O. Pastor and J. C. Molina. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer-Verlag New York, Inc., 2007.

[19] C. Pfister and C. Szyperski. Summary. In *Proceedings of the International Workshop on Component-Oriented Programming (WCOP96)*, 1997.

[20] T. Stahl, M. Völter, and K. Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2006.

[21] J. Steel and J.-M. Jezequel. On model typing. *Software and System Modeling*, 6(4):401–413, 2007.

[22] C. Szyperski. Component Software: Beyond Object-Oriented Programming, 1997.

[23] V. Vittorini, M. Iacono, N. Mazzocca, and G. Franceschinis. The osmosys approach to multi-formalism modeling of systems. *Software and System Modeling*, 3(1):68–81, 2004.

[24] T. Zhang, F. Jouault, J. Bézivin, and J. Zhao. A mde based approach for bridging formal models. In *TASE '08: Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 113–116, Washington, DC, USA, 2008. IEEE Computer Society.