# The software architecture of the OsMoSys Multisolution Framework

F. Moscato
Dip. di Informatica e Sistemistica
Univ. di Napoli Federico II
francesco.moscato@unina.it

F. Flammini
Dip. di Informatica e Sistemistica
Univ. di Napoli Federico II
frflammi@unina.it

G. Di Lorenzo
Dip. di Informatica e Sistemistica
Univ. di Napoli Federico II
giusy.dilorenzo@unina.it

V. Vittorini
Dip. di Informatica e Sistemistica
Univ. di Napoli Federico II
vittorin@unina.it

S. Marrone
Dip. di Ingegneria della Informazione
Seconda Univ. di Napoli
stefano.marrone@unina2.it

M. Iacono
Dip. di Studi Europei e Mediterranei
Seconda Univ. di Napoli
mauro.iacono@unina2.it

## ABSTRACT

The use of multi-formalism techniques is very appealing in modeling complex systems since they allow for building of complex models by integrating or composing sub-models specified by different formalisms. Hence, the most suitable formalism may be used according to the evaluation goals, the level of abstraction of the sub-models and the nature of the sub-systems. Each formalism is usually coupled with efficient solution methods, thus multi-solution approaches are needed to solve multi-formalism models whose analysis involves different techniques and tools. In this paper the software architecture of the OsMoSys Multi-solution Framework (OMF) is presented. OMF was born to provide the support needed to allow for loosely coupled cooperation among heterogeneous analysis techniques and tools, and automates the tasks that must be performed to solve complex multi-formalism models. OMF does not require that heterogeneous models are translated into a common formalism in order to be solved, nor that the available tools are modified to be integrated in the framework, but it achieves multi-solution by orchestration.

## 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, Performance, Reliability, Verification


*Valuetools* '07, October 23-25, 2007, Nantes, France


## Keywords

System Modeling, Multiformalism, Multisolution, Orchestration

The work of F.Moscato has been supported by the S.Co.P.E. project *http://www.scope.unina.it/progetto/default.aspx*

## 1. INTRODUCTION

The growing complexity of real world systems increases the need for effective design strategies to fulfill the requirements and reduce the development costs. An interesting approach to achieve this goal is moving resources from the verification and validation phases to the early stages of systems development in order to obtain better specifications, on which it is possible to investigate design properties. At this aim, the application of formal methods in the development cycle is advocated also by international standards. Nevertheless, the study of a complex system needs to cope with complexity and the heterogeneity of its subsystems. Complexity and heterogeneity can be faced through a divide-et-impera strategy, by using compositional modeling techniques, exploiting at best system decomposition in functionalities, subsystems and/or different aspects of the system itself. Multi-formalism multisolution approaches allow to operate on each aspect of the system by using the most appropriate modeling formalism and the best solution technique. Hence, they seem to be a flexible way to model complex systems, but require the availability of proper and powerful modeling methodologies and tools.

In this paper we introduce the software architecture of the OsMoSys Multi-solution Framework (OMF). OMF is part of the OsMoSys (Object-baSed multiformalism MOdeling of SYStems) research project, whose goal is the definition of a methodology and the realization of a software environment for the development and the analysis of multiformalism models. OMF is born to provide the support needed to a loosely coupled cooperation among heterogeneous analysis techniques and tools and to automate the tasks that must be performed to solve complex multiformalism models. ŞMulti-solutionŤ in OsMoSys means to solve a model - built by composing more sub-models - according to a well defined solution process which involves the execution of more solution or analysis tools (solvers). The execution order of the

solvers and the data dependencies among them are defined in the solution process that is described by means of a workflow language. In other words, OMF achieves multisolution by orchestration of solvers.

The paper is organized as follows. Related work is discussed in Section 2. Section 3 briefly describes the OsMoSys research project and places OMF in its context. The OMF software architecture is then presented in Section 4. In Section 5 an example is used to describe how a multiformalism model is solved and analyzed by means of the proposed architecture. Finally, Section 6 draws some concluding remarks.

## 2. RELATED WORK

The need for tools allowing the application of multiple solving strategies to models is well documented in literature.

In this Section the main approaches to multiformalism are briefly described, with particular emphasis on the solution strategy adopted, due to the scope of this work.

One of the first tools in the multiformalism field is SHARPE [18, 20] which focuses on performance and dependability evaluation. SHARPE supports both multiple solvers and multiformalism. It allows modular composition of models and each sub-model is solved by the appropriate solver. Solvers exchange results by expressing them in exponential polynomial distribution functions. In SHARPE the solving process is somehow guided by the modeler, interactively or via batch files. At the best of our knowledge, SHARPE does not natively support the extension of the number of formalisms or solvers it can use, nor needs a dedicated subsystem for result handling and presentation.

The DEDS toolbox [2] supports multiformalism and multiple solvers as well. One of the principal feature of the DEDS toolbox is its design for extension. This is obtained by the introduction of a high level (text-based) interface format (namely APNN, Abstract Petri Nets Notation), shared by all the components of the toolbox architecture. APNN is the basis for extensions because every solver that is compliant with it can be included in the framework. The language supports hierarchical models and constructs for exchanging results between levels, that are translated according to the applied solver.

The MODEST approach [4] (supported by the tool-suite MOTOR) is based on a single formalism for systems specification that is used to write models solved by multiple solvers. Hence, the language (that is an extended process algebra) has a non-trivial intrinsic complexity and each solver extracts a simpler model containing the needed information from the main model. This approach is oriented to keeping coherence when analyzing a same system specification by different points of view.

Möbius [8, 19] supports both multiformalism and multiple interacting solvers. The Möbius most elementary model is the Atomic Model, which is composed by Actions, State Variables and Properties. Adding the description of the desired reward variables (what is to be evaluated) a Reward Model is obtained. Models can be composed in order to exploit modularity and multiformalism, obtaining a Composed Model. A result is obtained as a computed solution to a reward variable. If a result has to be used for further computation, then it may capture the interaction between multiple Reward Models. A Connected Model is an ordered set of Reward Models and their solution methods. This requires that input parameters to some of the models depend on the results of other models in the set [8].

This paper focuses on the OsMoSys multisolution approach. Differently from other frameworks, which are based on "integrative" approaches, OsMoSys is based on orchestration, which is very advantageous in terms of flexibility and reuse. The models solved by means of OMF may be composed by sub-models expressed through different modeling languages, the tools used to implement the related solution techniques are orchestrated in order to perform the model analysis. The tools are wrapped so that they present the same interface to the orchestration process, but it is not necessary to modify them or add specific modules to convert the model definition to some particular description. The orchestration process is defined by means of an ad-hoc workflow language and it can be partially generated from the structure of the composed model and the information provided by the operators used to connect the sub-models, as explained in Section 4.

## 3. INTRODUCTION TO OSMOSYS

In this Section we summarize the main concepts which will be used in the following. OsMosys provides both a modeling methodology (OMM) and a software framework.

The *OsMoSys Modeling Methodology* (OMM) aims at the definition of a well formalized conceptual framework for the multi-formalism modeling promoting reuse and extensibility of complex models. OMM, first introduced in [22], is based on object orientation concepts and metamodeling. In OMM different modeling formalisms, (e.g. Petri Net, Fault Tree, Process Algebra, etc.) may be defined using a metalanguage, named *Metaformalism*. In the OsMoSys terminology a formalism defined by means of the *Metaformalism* is a *Model Metaclass* (Metaclass for short) and new Metaclasses can be easily introduced, or defined by inheritance from existing ones (e.g. a Generalized Stochastic Petri Net Metaclass may be obtained by inheriting from a Petri Net Metaclass).

The Metaclass defines the set of element types belonging to the related formalism (e.g. nodes, edges, actions, etc.) that a model may include. Each element type has a name and one or more *properties* that are used to specify the attributes of the element. For example if we consider the Petri Net (PN) formalism, an element type of a PN Metaclass may have name *Place* and a property *Tokens* that represents the initial marking of a place.

A *Model Class* describes a model (compliant with a given Metaclass) that needs to be instantiated to obtain a *Model Object*. Hence, a *Model Class* is a model which specifies some parameters that must be defined before the model is used. A *Model Class* may have an interface to connect its Model Objects with the external environment. A *composed model* is built by connecting Model Objects (sub-models) through proper operators. The Model Classes that may be used to build the composed model, the topology of the composition and the operators allowed for that composition are defined by a special formalism named *Bridge*. OMM distinguishes between explicit and implicit multi-formalism. *Explicit* multi-formalism is visible at the user level: the user may build a composed model by explicitly using Model Classes compliant with different Metaclasses. *Implicit* multi-formalism is a form of multi-formalism that is not visible at the user level, since it is only exploited in order to solve the models.

OMM has been recently extended to introduce Model Interface Polymorphism [16], since the ultimate goal of OsMoSys is not only to promote model reuse, but to improve the overall modeling approach to the analysis of complex system by allowing for dynamic construction of models. In particular, Interface Polymorphism allows to define parts of a multi-formalism model on-the-fly, i.e. while solving the model.

The very first steps of OsMoSys are described in [12]. OMM has been applied to the definition of Repairable Fault Trees [7] (a case of implicit multi-formalism) and to the development of different models, for example in [11] (contact center scenarios) and [9] (evaluation of design choices for critical control systems). The example presented in [14] will be used here to show how the solution process works in practice.

The development of a model according to OMM is supported by the DrawNET modeling system [6].

The analysis of complex OMM models can be performed by means of the OsMoSys Multi-solution Framework (OMF), whose complete software architecture is described in this paper.

# 4. SOFTWARE ARCHITECTURE OF THE OSMOSYS FRAMEWORK

Metaclasses, Model Classes and Model Objects are described in OMF by a family of XML based languages [13, 17]. Once a model has been developed, it is expressed by means of the Model Definition Language (MDL) that is used for both Model Classes and Model Objects. To solve a model a *Query* must be defined on the model itself which contains a specification of the performance indices or measures that must be evaluated. A Query is expressed by means of the Model Query Language (MQL). Then a solution process is generated. The solution process specifies a workflow [1, 23], i.e. it describes which tasks must be performed to solve the model, how they are structured, who performs them, what their relative order is, how they are synchronized, how information flows to support them. Hence, the solution process is expressed by means of a workflow language, the Solution Process Definition Language (SPDL) introduced in [17]. A solution process is then executed by a workflow engine. The results obtained must be collected and processed - if it is necessary- to produce the answer to the Query defined on the model. The answer to a Query is expressed by means of the ReSult Language (RSL).

Fig. 1 shows the OMF architecture and the graphical user interfaces (GUIs) which are used to interact with OMF. The GUIs (represented by the three rectangles in the pale-gray box on top of Fig. 1) do not strictly belong to the OMF architecture (depicted in the big dark-gray box). The *Drawing GUI* is used to develop the models and generate their MDL definitions. At the moment the model development is supported by the DrawNET modeling system [6], as mentioned in Section 3. The *SPD GUI* aids the editing of a solution process that must be written in SPDL. The *Configuration GUI* is a graphical console used to configure and manage OMF. The OMF architecture mainly consists of the following layers:

1. the SPDL Compiler;
2. the OsMoSys Core;
3. the Solvers/Applications layer;
4. the Repositories/DB layer.

The SPDL Compiler partially automates the generation of a solution process, at this aim the compiler uses the model definition written in MDL and the information about the current OMF configuration.

The second layer contains the OMF components used to execute the solution process. These components are able to:

- handle queries and results (Results Manager);
- instantiate model classes (Instancer);
- enact the solution process (Worklow Engine).

The third layer contains solvers and tools used to solve sub-models. At this layer other tools, like Möbius ([8, 19]), SHARPE ([18]), UPPAAL ([3]) or GreatSPN([5]) can be used to solve submodels. In order to use existing tools it is necessary to write a wrapper, called *Adapter*. An Adapter is used to manage input/output formats, invoke a solver and handle results. If allowed by the solver, the Adapter can also interact with it during its execution. For example, this feature can be used in order to manage the state-space evolution of the model during its solution.

The relationships among all the entities involved in the analysis of a model are described and handled by the Repository/DB layer. In particular, it is necessary to store and manage information about the solvers and their Adapters, about the relationships among formalisms, models and measures or performance indices which solvers are able to analyze/evaluate, about the associations between models and queries, about (sub)models and their related results: all the information needed to define multiformalism models and to enact solution processes, or that must be stored for re-use purposes, are contained in the OMF Data Bases.

The components of OMF are described in more details in the remainder of this Section.

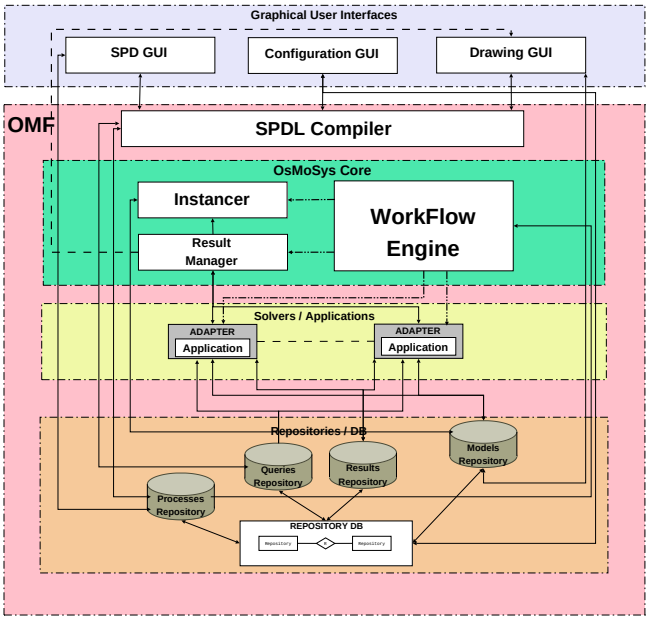


**Figure 1: OsMoSys Architecture**

## 4.1 Instancer

OsMoSys defines a language called Model Definition Language (*MDL*) to write Model Classes and Model Objects specifications, as explained before. Let $m^c \in MDL$ denote a Model Class written in MDL. The Instancer performs the operation:

$$instance : m^c, par_{m^c} \rightarrow m^o \in MDL$$

that is, the Instancer transforms a Model Class specification into a Model Object specification $m^o$ (still written in MDL). The Model Object is instantiated with the parameters $par_{m^c}$. These parameters are the values that have to be assigned to some element properties in the Model Class in order to produce a fully defined model.

If the value of the parameters are known at solution time, the *Results Manager* produces the $par_{m^c}$ information by elaborating the results retrieved from the analysis of some sub-models.

Hence, the Instancer can be invoked:

- by the modeler, to instantiate Model Classes stored in the Models Repository;
- by the workflow engine, to instantiate Model Classes using results from the anlysis of sub-models previously solved during the solution process enactment.

The first usage of the Instancer enables model reuse and produces solvable Model Objects. The second one is useful to implement composition semantics.

Inputs to the Instancer are models specifications that must be instantiated, and the value of the parameters needed for the instantiation. Outputs of the instantiation are the instantiated model specifications.

## 4.2 Solvers

Solvers are external tools or software applications used to analyze and solve models. For example the tool *newSO* of the *GreatSPN* package [5] can be used to perform the steady state analysis of a Generalized Stochastic Petri Net (GSPN) model.

Let $m_s$ denote a model to be solved by an external tool. The solver uses its input format for the model description. Let $Smf$ denote the description language used by the solver. A *solver* performs the operation:

$$solver\text{: } m_s \in Smf \rightarrow r_s \in Srf$$

where $r_s$ is the result of the analysis produced by the solver. It is expressed by means of an output format. $Srf$ denotes the description language by which the results produced by the solver are expressed. It is important to underline that each solver usually use its own input/output format. Input/Output solvers's data are wrapped in the OMF languages by the Adapters.

## 4.3 Adapters

Adapters are software modules that directly interact with solvers. They are wrappers whose goal is to provide the solvers with a common interface to the solution process and the workflow engine. The tasks executed by an Adapter mainly include: the translation of the solver input/output into the OMF proper language, and the interaction with the solver during the solution process execution. An Adapter also invokes the execution of the solver. Then it waits for the end of the execution in order to retrieve the results of the analysis. An Adapter can also interact with the solver during its execution if allowed by the architecture or the implementation of the solver itself. In this case, if the solution of the model is performed by analyzing its state space, and if the state space representation can be accessed, the Adapter is able to handle this representation during the model analysis or simulation.

Hence, an Adapter may perform the following operations:

(1) $adapter : m^o \in MDL \rightarrow m_s \in Smf$

(2) $adapter : r_s = solver(m_s)$

(3) $adapter : r_s \in Srf \rightarrow r_s^{OMF} \in RSL$

where $m^o$ is a Model Object expressed by MDL, $m_s$ is the same model expressed by the solver format, $r_s$ is the result obtained by executing the solver on the model $m_s$ and $r_s^{OMF}$ is $r_s$ expressed by the OMF format.

The first operation is needed to provide the solver with the right input, the second operation is the invocation of the solver, and the third is necessary in order to allow the Results Manager to retrieve results from a solved model, without writing a results manager for each solver.

The Adapter has to store in the $Repository/DB$ the exact correspondence between the elements of the Model Objects $m^o$ and the elements of the formalism by which $m_s$ is described.

In order to make easier the Adapters development, OMF provides a library (the *Adapter Interface* library) which contains pre-compiled functions to:

- manage OMF languages, models, queries and results;
- define a common interface to solvers's state space representations;
- manage communications with solver (through file exchange, pipe, socket and message);
- define proper interfaces to solvers's functionalities.

These functions can be used in order to develop wrappers for solvers able to interact with other OMF components.

## 4.4 Results Manager

The Results Manager (RM) is the OMF software module in charge of processing the results obtained by the activation of Adapters and producing the answers to the queries defined on the models. Indeed, the Results Manager operates on results translated to the OMF format and stored in the *ResultsRepository* by Adapters.

Nevertheless a query always asks for the evaluation of measures or performance indices over a model, we distinguish two kinds of queries: a) an *End-User Query* is the query over a model solved by means of the enactment of a solution process; b) an *Intermediate Query* is a query generated by the SPDL Compiler and used to define the solution process. Thus, more Intermediate Queries may be produced to answer to one End-User Query. The simplest case is the generation of an Intermediate Query over a submodel, required to cause the invocation of an Adapter (and the execution of the associated solver). The answer to these

Intermediate Queries is mainly used for instantiation purposes (to set the values of Model Class parameters during the solution process as explained in Section 4.1).

A more complex case is the generation of Intermediate Queries when the End-User Query refers to a model which exploits *Implicit Multiformalism*. In this case, the results requested by the End-User Query may be related to models that are generated during the solution process, so that the relationship between the Intermediate Queries and the End-User Query is not immediate and there is not a direct relation among the results obtained during the solution process and the answer to the End-User Query. Proper functions have to be given in order to define the relationship between queries and results. These functions are called *Translation Functions*.

Let $mq^u \in MQL$ denote an End-User Query, a Translation Function $Q_T$ must be given to express $mq^u$ by means of $r$ Intermediate Queries:

$$Q_T : mq^u \in MQL \rightarrow mq_1^s, \cdots, \ mq_r^s \ \in MQL$$

where $mq_k^s$ ($k \in \{1, ..r\}$) are written in $MQL$ but refer to sub-models that are not explicitly part of the model referred by $mq^u$. A Translation Function $R_T$ must be given to calculate the result asked by the End-User Query from the answers to the $r$ Intermediate Queries. Let $rs_k^s \in RSL$ ($k \in \{1, ..r\}$) denote the results obtained during the solution process to answer to the queries $mq_k^s$:

$$R_T : rs_1^s, \cdots, \ rs_r^s \in RSL \rightarrow rs^u \in RSL$$

The Translations Functions are defined off-line and used by the Results Manager. They are associated to the available Bridge formalisms and stored in the *Repository*.

## 4.5 Workflow Engine

The Workflow Engine (WFE) is the core of the solving infrastructure of OMF: it orchestrates the other OMF components in order to enact the solution processes used to solve models. Workflow languages are the best candidates for describing complex orchestration patterns [21] and for this reason SPDL is a workflow language [17] developed to describe the OMF solution processes, that are (ad-hoc) workflow processes [23].

Complex control flow path and data path can be defined by SPDL. The invocation of Adapters or other software modules is represented by an *Activity*. *Transitions* link activities and define the execution order. Proper conditions may be defined in order to allow parallel, sequential or conditional execution of activities. Loops can also be defined, by using proper activities. Each activity is performed by a *Participant* which is an elaborating node able to execute Adapters or other software modules. Since a software module may be replicated on more participants, the WFE is also able to perform a dynamic load balancing when invoking Adapters and other applications.

Hence, in order to manage the solution process, the workflow engine performs the following tasks:

- Instantiates the applications needed to perform the solution process activities;
- Performs a dynamic load balancing on participant nodes;
- Performs routing of data needed by the applications to accomplish their tasks.

Proper scheduling strategies are implemented in the WFE kernel, that is described in [17].

## 4.6 SPDL Compiler

The SPDL compiler allows for building solution processes from the model definitions and from the End-User Queries.

Multi-formalism models are described by using particular formalisms called *Bridges* as explained in Section 1.

Sub-models and *operators* are elements that belong to Bridge Model Classes. *Operators* are used to connect submodels and carry information about the composition semantics.

Operators have been introduced in [15] and the discussion of this topic is out of the scope of the paper. In the following the operators will be briefly described in order to explain how they are used by the SPDL Compiler in order to support the automatic generation of a solution process.

OMM distinguishes two kinds of operators: operators that imply models manipulation and operators which require the evaluation of results from submodels in order to instantiate or modify other sub-models during the enactment of a solution process.

The semantics of the operators related to models manipulation strictly depends on the formalisms used to describe sub-models, the operators related to the evaluation of results are more general and they can be applied to all sub-models. Sub-models linked by edges to operators are their operands. Depending on edges orientation it is possible to distinguish input and output operands.

The analysis of the graph by which the composed models are defined produce information useful to generate the solution processes.

From the graph structure of a composed model it is possible to identify:

- sequences of activities that have to be enacted by the WFE;
- subprocesses that can be executed in parallel;
- join and split points in the process control flow.

The SPDL Compiler is divided into two main components: a *front-end* and a *back-end*. The front-end parses the Bridge graph structure producing an intermediate code that can be used by the back-end in order to define a solution process.

The pseudo-code of the main front-end analysis procedure is reported below:

```
1.Bridge-Graph subGr;
2.Bridge-Graph ordSubGr;
3.list ordOperators;
4.list ordOperands;
5.// orders subGr byusing a topological depth first
6.// ordering algorithm
7.ordSubGr := DepthFirstOrder(subGr);
8.//separates the (ordered) lists of operators
9.//and operands
10.ordOperators := operatorsIn(ordSubGr);
11.ordOperands :=operandsIn(ordSubGr);
12.for (each Operand)
13.begin
14.  identify pre/post conditions;
15.end
16.identify sequences
17.identify parallels
```

```
18.translate operators in sequences of micro-operations
19.produce front-end output.
```

The graph that represents the model to analyze is denoted *subGr* (line 1).

First of all *subGr* is ordered by using a depth first topological ordering algorithm (line 7) and then operators and operands are divided into two lists maintaining this order (lines 10,11). The ordering brings a definition of pre and post conditions over operators.

An operator pre-condition is a condition that has to be verified in order to allow the enactment of actions related to the operator semantics. An operator post-condition defines which are the submodels that can be analyzed after the enactment of a such an action.

The pre and post conditions are identified (loop on line from 12 to 15) and then analyzed by the compiler. Depending on pre and post conditions, activities that can be executed in sequence or in parallel are identified (lines 16,17). In addition join and split points in the control flow are defined.

Notice that if an operator $Op_1$ has pre-condition $pre_1$ and post-condition $post_1$, and if $Op_2$ has pre-condition $pre_2$ and post-condition $post_2$, and if $post_1 = pre_2$, than $Op_2$ has also $pre_1$ as pre-condition. We address briefly this property as *condition chaining*.

The operator semantics is translated into a sequence of solution process activities (called *micro-operations*). These activities depend on source (input) and destination (output) operands of an operator. In the case of operators related to model manipulation, the micro-operations depend on operands structures and formalisms. If the models requires the exploitation of result-related operator, the micro-operations to enact during the solution process involve directly the execution of the OMF core components. The more common micro-operations (MO) that the Compiler can manage are the following:

- **Compute**(property on a model);
- **Assign** (source, destination);
- **Evaluate** (expression);
- **Retrieve** (property on a model).

The Compute MO allows for the evaluation of a property on a model by solving it. It involves the definition of a proper query on a (sub)model and the execution of a proper Adapter in order to retrieve the value of the property. The obtained result is then stored in the proper OMF *Repository*.

The Assign MO allows a given value (source) to be assigned as property of a given (sub)model element (destination). This involves the execution of the Results Manager if the source was a previously retrieved result, and the execution of the Instancer on the proper model to build a Model Object.

The Evaluate MO is used to evaluate expressions on given elements (for example variables, other results etc). The Results Manager parses the expression evaluating the results. If expression variables are model properties or results, it retrieves their values from model definitions and results in the *Repository*.

The Retrieve MO is similar to Compute with the difference that no solution is requested. Adapters will not be invoked and only the Results Manager is involved in this MO.

The above MO can be nested to build more complex operators. For example the following micro-code defines the assignment: $M2.e2.p2 = 1/(M1.e1.p1)$, where $M1$ is a model object to solve, $M2$ is a model class to instatiate, $e1$ and $e2$ are respectively elements of $M1$ and $M2$ and finally $p2$ is a property of the element $e2$ and $p1$ is a property of the element $e1$. The value of $p1$ is not contained in the $M1$ definition but can be computed by solving the model. The value of $p2$ is determined by applying the previous assignement.

```
<ASSIGN>
 <LEFT> "M2.e2.p2" </LEFT>
 <RIGHT>
   <EVALUATE> 1/<COMPUTE Par="M1.e2.p1"/> </EVALUATE>
 </RIGHT>
</ASSIGN>
```

This kind of operator will be used in Section 5.

The output of the compiler front-end is an XML document. It contains both information about sequential and parallel paths in the solution process control flow, and the sequences of MOs to be used to generate the solution process.

The compiler back-end will get this output in order to substitute the right OMF component invocation into the solution process, translating the front-end representation into SPDL. The *Repository* will be used in order to choose the right Adapter and options to be activated during the solution process.

## 4.7 The User View

Fig. 2 shows the OMF users point of view. There exist three main types of users:

1. users wishing to integrate existing tools or solvers into the framework (actor *Tools Integrator*);
2. users writing solution processes (actor *Solution Writer*);
3. users that only use the framework choosing models and solution processes from libraries (actor *Model User*).

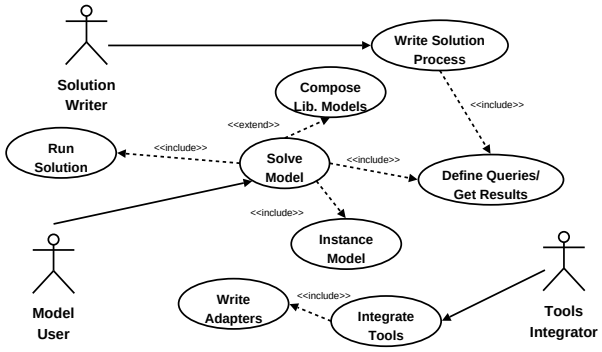


**Figure 2: OsMoSys User View**

The *Tool Integrator* integrates existing tools into the framework. As described in Section 4.3, to accomplish this task an Adapter must be written by using the Adapter Interface

library. The other users can use Adapters and tools in the solution process they wish to enact.
The *Solution Writer* writes solution processes for a certain set of model classes. The OMF Compiler is able to write solution processes automatically, but the following situations may happen:

- an user want to define a solution process without invoking compiler services.
- more solvers are "adapted" in the OMF which can enact a given COMPUTE MO.
- complex queries are defined on the user model and operators are not directly related to that queries.

In such cases, the Solution Writer must to be able to create (in the first case) a solution process or to modify (for other cases) a solution process produced by the compiler. In the second case, the actor modifies adapter invocations directly in the SPDL code produced by the compiler, while in the last case usually it must be able to modify the structure of the solution process introducing further steps to obtain the needed results.

The Solution Writer uses the SPD GUI to write the solution process and to define queries needed to implement the bridge composition semantics.
The last actor is the *Model User*, who has the main purpose to solve a model by choosing a solution process from the library within the ones defined for that model (*Solve Model*). To accomplish this task, all model classes have to be instantiated (*Instantiated Model*). Then queries have to be defined on the model (*Define Queries/Get Results*) in order to retrieve results after the execution of the solution process (*Run Solution*) enacted by the *WFE*. The Model User mainly interacts with the Drawing GUI.

We remember that two kinds of queries can be defined in the OMM:

- End-User Queries: they are defined by the user to specify results he/she wants to evaluate from the model solution.
- Intermediate QUeries: they are used to define and properly enact the solution process needed to compute the results defined in the previous point. These queries are generated by the back-end of the OMF compiler when a COMPUTE MO is defined in the front-end output.

## 5. A MULTI-SOLUTION EXAMPLE

In this Section we exemplify the use of OMF. The starting point is a multiformalism model of a simple RAID 5 system described in [14]. The focus here is not on the construction of the model, but on the generation of the solution process and the role of the operators.

The system consists of two controllers (one is integrated into the disk array, the other issues requests to the first) and three disks. Data can be written in full stripes (the most of the cases, according to caching mechanisms) or half stripes, and are always read in full stripes. Integrity of the system relies on disks: if no disk is damaged the system is in the *Ok* state, if one disk is damaged the system is in the *Degraded* state, if two or all of the disks are damaged the system is in the *Dead* state (it is not working). A performability model of this system is built to evaluate the mean response time of read and write operations (RWMRT) while the system is working. The model is obtained by composing seven sub-models.

Two GSPN models are used to evaluate the throughput of read and write operations when the system is working in the *Ok* or *Degraded* state (they are named $GSPN_{OK}$ and $GSPN_{DEGR}$ in the following). The throughput is associated to a transition of the GSPN models named *Sync*.

Simple queues are used to evaluate the queuing effects on read and write operations. They are described by means of the Queueing Network (QN) formalism. The mean of the service rate distributions of $QN_{OK}$ and $QN_{DEGR}$ are the inverse of the throughput of the *Sync* transition of $GSPN_{OK}$ and $GSPN_{DEGR}$ respectively (in the hypothesis that the queues can be approximated by M/M/1 queues).

$QN_{CONTR}$ is the QN that describes the behavior of the disk array controller; to obtain the RWMRT of the system, the arrival distribution of $QN_{CONTR}$ takes into account the probability that the system is in the *Degraded* or *Dead* state: two Fault Tree models ($FT_{DEGR}$ and $FT_{DEAD}$) are used to evaluate these probabilities that are associated to the Top Events of the FT models, named $TE_{DEGR}$ and $TE_{DEAD}$.

A Bridge formalism is used to build a composed model able to contain Model Classes compliant with GSPN, QN and FT Metaclasses and the proper composition operators.

The composed model of the RAID system is shown in Fig. 3, where squares encapsulate Model Classes and rhombuses represent operators linked to the model interfaces by means of edges.

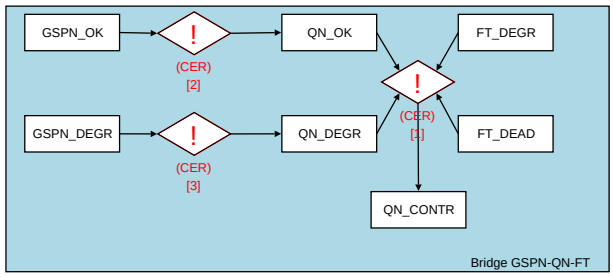


**Figure 3: The case-study model**

The Model Classes $GSPN_{OK}$, $GSPN_{DEGR}$, $FT_{DEGR}$ and $FT_{DEAD}$ are then instantiated (the parameters of GSPN models are the operations rates obtained through real measures from a RAID system and the parameters of the FT models are the mean time between failures of the disks which are given by their specification). Nevertheless, the QN Model Classes cannot be fully instantiated, since the value of some parameters will be known only during the solution process: the semantics of this composition requires exchanges of information (results) between the sub-models in order to obtain the automated solution of the RAID model. The Query defined on this model asks for the evaluation of the RWMRT of the system, that is obtained by evaluating the sojourn time distribution of the $QN_{CONTR}$ model. Hence, the main steps to solve the models are the followings:

1. The $GSPN_{OK}$ model is solved to compute the *Sync* transition Throughput.
2. The inverse of this value is evaluated and assigned

to the service rate of the server in the $QN_{OK}$ model ($QN_{OK}$ is now fully instantiated).

3. The $GSPN_{DEGR}$ model is solved to compute the *Sync* transition Throughput.
4. The inverse of this value is evaluated and assigned to the service time of the server in $QN_{DEGR}$ model. ($QN_{DEGR}$ is now fully instantiated).
5. The $FT_{DEGR}$ model is solved to compute the $TE_{DEGR}$ probability.
6. The $FT_{DEAD}$ model is solved to compute the $TE_{DEAD}$ probability.
7. The $QN_{OK}$ model is solved to compute the mean sojourn time.
8. The $QN_{DEGR}$ model is solved to compute the mean sojourn time.
9. The $QN_{OK}$ mean sojourn time is weighted by the 1-($TE_{DEGR}$+$TE_{DEAD}$) probability.
10. The $QN_{DEGR}$ mean sojourn time is weighted by the $TE_{DEGR}$ probability.
11. The sum of these weighted mean sojourn times is evaluated and assigned to the arrivals mean rate of the $QN_{CONTR}$ model.
12. The $QN_{CONTR}$ model is solved in order to compute the RWMRT.

The semantics of the **CER** operator in Fig. 3 is the following: it requires the computation of values of some elements parameters of some models (Source Models) which are linked to it by incoming edges. The computed results are manipulated in order to be used as instantiation information for other submodels (Destination Models) parameters. The destination models are linked to the operator by outgoing edges. In brief CER operator needs three MO to be enacted, an ASSIGN, an EVALUATE and a COMPUTE, in the following order :

```
<ASSIGN>
 <LEFT> "Destination Model parameter" </LEFT>
 <RIGHT>
   <EVALUATE>
    f( <COMPUTE Par="Source Model parameter"/>)
   </EVALUATE>
 </RIGHT>
</ASSIGN>
```

As for the step previously described, step **1** requires an ASSIGN to be performed, while step **2** requires to perform a chaining of the COMPUTE and EVALUATE operators.

The front-end intermediate code generated by the compiler for the case study is reported below:

```
<SEQUENTIAL>
  <PARALLEL>

    <SEQUENTIAL>
      <ASSIGN>
        <LEFT> "QN_OK.serv.rate" </LEFT>
        <RIGHT>
          <EVALUATE>
            1 /<COMPUTE Par="GSPN_OK.Sync.Throughput" />
          </EVALUATE>
        </RIGHT>
      </ASSIGN>
    </SEQUENTIAL>

    <SEQUENTIAL>
      <ASSIGN>
        <LEFT> "QN_Degr.serv.rate" </LEFT>
          <RIGHT>
            <EVALUATE>
              1 /<COMPUTE Par="GSPN_Degr.Sync.Throughput"/>
            </EVALUATE>
          </RIGHT>
      </ASSIGN>
    </SEQUENTIAL>

  </PARALLEL>

  <SEQUENTIAL>

    <ASSIGN>
      <LEFT> "QN_CONTR.arrivals.rate" </LEFT>
      <RIGHT>
        <EVALUATE>
          (1-(<COMPUTE Par="FT_Degr.TEdegr.Prob"/>+
          <COMPUTE Par="FT_Dead.TEdead.Prob"/>)*
          <COMPUTE Par="QN_OK.sojourn.mean"/> ) +
          (<COMPUTE Par="FT_Degr.TEdegr.Prob"/>)*
          <COMPUTE Par="QN_Degr.sojourn.mean"/> )
        </EVALAUTE>
      </RIGHT>
    </ASSIGN>

    <COMPUTE Par="QN_CONTR.sojourn.mean" />
  </SEQUENTIAL>

</SEQUENTIAL>
```

The algorithm described in section 4 is applied to the bridge model in Fig.3 as described in the following. Submodels and operators are the nodes of the graph that are ordered by applying the $DepthFirstOrder$ algorithm. Then pre and post conditions are identified. The instantiation of the $QN_{CONTR}$ submodel is a post-condition for the $CER$[1] operator and the solution of $QN_{OK}$, $QN_{DEGR}$, $FT_{DEGR}$, and $FT_{DEAD}$ submodels are pre-conditions of the same operator. Post-condition and pre-condition for the $CER$[2] operator are respectively the instantiation of the $QN_{OK}$ submodel and the solution of the $GSPN_{OK}$. Finally the instantiation of the $QN_{DEGR}$ submodel and the solution of the $GSPN_{DEGR}$ are respectively the post-condition and the pre-condition of the $CER$[3] operator. Every solution of a submodel has its instatiation as pre-condition. Notice that the $GSPN_{OK}$, $GSPN_{DEGR}$, $FT_{DEGR}$ and $FT_{DEAD}$ submodels are model objects in the bridge model definition. The condition chaining property states that also the solutions of $GSPN_{OK}$ and $GSPN_{DEGR}$ submodels have to be considered as pre-conditions of the $CER$[3] operator.

$CER$[1] can be exploited only after the enactment of $CER$[2] and $CER$[3] operators. In addition, these two last operators do not have any common pre-condition and can be enacted in parallel. For this reason the front-end output contains a solution process skeleton which comprises the enactment of $CER$[2] and $CER$[3] in parallel followed (in a sequence definition) by the exploitation of the $CER$[1] operator.

The back-end translates the previous code into an SPDL representation of the solution process. Notice that further optimization can be exploited in this step: all COMPUTEs

related to the FT models in the last SEQUENTIAL step can be executed concurrently with the GSPN models solutions; in addition the sojourn mean time in the $QN_{DEGR}$ queue can be evaluated once in the last EVALUATE section.

The SPDL code defines a solution process workflow which is depicted in Fig. 4.

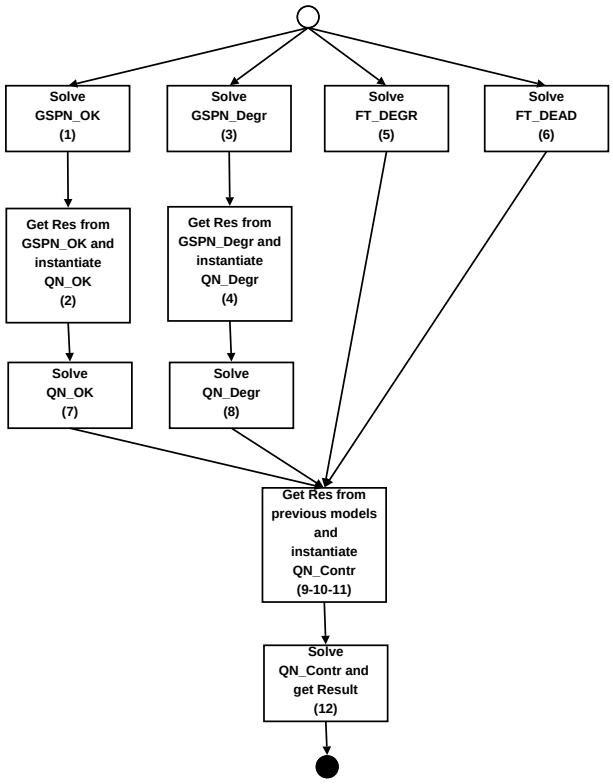


**Figure 4: The case-study solution process**

Each **Solve** *Model* phase is related to a COMPUTE. It involves the execution of the proper Adapter to solve the *Model* and retrieve results from it. The result to retrieve is specified in a proper query.

Each **Get Res** from *Model1* and **instantiate** *Model2* is related to ASSIGNs. Eventually some the results can be manipulated by some expressions (when an EVALUATE MO is defined in the front-end code). This phase is performed by the Result Manager that is able to retrieve results from the proper OMF repository and to evaluate some expressions using results values. In addition the Result Manager is able to produce instantiation information for submodels that have to be instantiated later during solution process enactment.

Notice that in Fig. 4 edges outgoing the same node represent split points in the solution process, while edges incoming in the same node represent join points.

## 6. CONCLUSIONS

In this paper the architecture of the OsMoSys Multi-solution Framework (OMF) has been presented. The framework is being developed in the context of the OsMoSys research project whose aim is the definitions of theoretical means and CASE tools to build multi-formalism models of complex systems, and to promote formal modeling in industrial settings. Hence, OMF provides the mechanisms to analyze and solve multi-formalism (composed) models, built according to the OsMoSys Modeling Methodology, i.e. according to an object oriented approach.

The framework supports flexible solution processes and allows for the integration of heterogeneous external solvers (or other software applications) at low development costs, simply implementing the Adapter Interface provided by the framework.

The OMF can be used by users expert in formal modeling but it can be also used to realize COTS (Commercial off-the Shelf) strategies in formal models development, since it may provide libraries of models and solution processes ready to be instantiated and used.

At the moment the framework is almost entirely implemented. The Compiler module, which has been recently introduced in OsMoSys, is being developed, and the same holds for the functionalities of the Instancer which allow to support the interface polymorphism. Nevertheless, the overall framework architecture is stable.

Future research work on the methodology is also necessary to fully define different classes of composition operators and extend the OsMoSys Modeling Methodology (OMM) with the concepts of template models and behaviour inheritance.

Further efforts will be focused on exploiting the OMF for solving multi-formalism models in the field of dependability and security of real-world complex systems and infrastructures, like the model presented in [10] which addresses the analysis of critical railway controllers.

## 7. REFERENCES


[1] Allen, R. 2001. Workflow: An Introduction. Extracted from *The Workflow Handbook 2001*, Workflow Management Coalition, http://www.wfmc.org/standards/docs.htm

[2] Bause, F., Buchholz, P., and Kemper, P. 1998. A Toolbox for Funtional and Quantitative Analysis of DEDS. *Proc. 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'98)*, 356–359.

[3] Behrmann, G., David, A., Larsen, K. G., Möller, O., Pettersson, P., and Yi, W. 2001. Uppaal - Present and Future. *Proc. of 40th IEEE Conference on Decision and Control (CDC'01)*, 2881-2886.

[4] Bohnenkamp, H., Hermanns, H., Katoen, J. P., and Klaren, R. 2003. The Modest Modeling Tool and Its Implementation. *Computer Performance: Modelling Techniques and Tools - Tools for Evaluation of Stochastic Models*, LNCS 2794, 116–133. Springer-Verlag.

[5] Chiola, G., Franceschinis, G., Gaeta, R., and Gribaudo, M. 1995. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation, special issue on Performance Modeling Tools 24*, 1&2 (November), 47–68. Elsevier.

[6] Codetta-Raiteri, D., Franceschinis, G., and Gribaudo, M. 2006. Defining formalisms and models in the Draw-Net Modeling System. *Proc. of the Fourth Int. Workshop on Modelling of Objects, Components and Agents (MOCA06)*, 123–144.

[7] Codetta Raiteri, D., Franceschinis, G., Iacono, M., and Vittorini, V. 2004. Repairable Fault

Tree for the Automatic Evaluation of Repair Policies, *Proc. of the Performance and Dependability Symposium*, 659–668.

[8] Deavours, D., Clark, G., Courtney, T., Daly, D., Derisavi, S., Doyle, J. M., Sanders, W. H., and Webster P. G. 2002. The Möbius Framework and its Implementation. *IEEE Transactions on Software Engineering 28*, 10, 956-Ű969.

[9] Flammini, F., Iacono, M., Marrone, S., and Mazzocca, N. 2005. Using Repairable Fault Trees for the evaluation of design choices for critical repairable systems. *Proc. 9th IEEE Symposium on High Assurance Systems Engineering (HASE2005)*, 163–172.

[10] Flammini, F., Marrone, S., Mazzocca, N., and Vittorini, V. 2006. Modelling System Reliability Aspects of ERTMS/ETCS by Fault Trees and Bayesian Networks. *Safety and Reliability for Managing Risk: Proceedings of the 15th European Safety and Reliability Conference (ESREL'06)*, 2675–2683.

[11] Franceschinis, G., Gribaudo, M., Iacono, M., Marrone, S., Mazzocca, N. and Vittorini, V. 2004. Compositional modeling of complex systems: contact center scenarios in OsMoSys. *Proc. of Applications and Theory of Petri Nets 2004, 25th international conference (ATPN'04)*, LNCS 3099, 177–196. Springer-Verlag.

[12] Franceschinis, G., Gribaudo, M., Iacono, M., Mazzocca, N., and Vittorini, V. 2002. Towards an object based multi-formalism multi-solution modeling approach. *Proc. of the 2nd Workshop on Modelling of Objects, Components and Agents (MOCA'02)*.

[13] Gribaudo, M., Iacono, M., Mazzocca, N., and Vittorini, V. 2003. The OsMoSys/DrawNET XE! Languages System: A Novel Infrastructure For Multi-formalism Object-Oriented Modelling. *Proc. 15th European Simulation Symposium and Exhibition*.

[14] Gribaudo, M., Moscato, F., Mazzocca, N. and Vittorini, V. 2005. Multisolution of Complex Performability Models in the OsMoSys/DrawNET Framework. *Proc. 2nd International Conference on the Quantitative Evaluation of Systems (QEST'05)*, 85–94.

[15] Moscato, F. 2005. *Multisolution of Multiformalism Models: Formal Specification of the OsMoSys Framework*, Ph.D. Thesis, Second University of Naples, Department of Information Engineering.

[16] Moscato, F., Gribaudo, M., Franceschinis, G., Mazzocca, N., and Vittorini, V. 2007. Introducing interface polymorphism in multi-formalism modeling. Submitted to *Journal of Software and System Modeling*. Springer.

[17] Moscato, F., Mazzocca, N., and Vittorini, V. 2004. Workflow Principles Applied to Multi-Solution Analysis of Dependable Distributed Systems. *Proc. of the 12th Euromicro Conf. on Parallel, Distributed and Network-based Processing*, 134–141.

[18] Sahner, R. A., Trivedi, K. S., and Puliafito, A. 1996. *Performance and Reliability Analysis of Computer Systems - An Example-based Approach Using the SHARPE Software Package*. Kluwer Academic Publisher.

[19] Sanders, W. H., Courtney, T., Deavours, D., Daly, D., Derisavi, S., and Lam, V. 2003. Multiformalism and Multisolution-method Modeling Frameworks: The Möbius Approach. *Proc. of the Symposium on Performance Evaluation - Stories and Perspectives*, 241–256.

[20] Trivedi, K. S. 2002. SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator. *Proc. of the Symposium on Dependable Systems & Networks (DSN'02)*, 544.

[21] van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B. and Barros, A. P. 2003. Workflow Patterns. *Distributed and Parallel Databases 14*, 3 (July), 5–51.

[22] Vittorini, V., Iacono, M., Mazzocca, N., and Franceschinis, G. 2004. The OsMoSys approach to multi-formalism modeling of systems. *Journal of Software and System Modeling 3*, 1 (March), 68-Ú81. Springer.

[23] WFMC Coalition. 1999. *Workflow Management Coalition Terminology and Glossary*, WFMC-TC-1011, http://wfmc.org.