Defining formalisms and models in the Draw-Net Modelling System

Daniele Codetta-Raiteri[1], Giuliana Franceschinis[1], and Marco Gribaudo[2]

[1] Dipartimento di Informatica, Università del Piemonte Orientale, Via Bellini 25/G, 15100 Alessandria, Italy. {raiteri, giuliana}@mfn.unipmn.it
[2] Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy. marcog@di.unito.it

Summary. This paper presents the *Draw-Net Modelling System* (DMS), a framework for the design and the solution of models expressed in any (graph based) formalism, including the possibility of representing complex models by means of multi-formalism and analyzing them by exploiting different solution modules. In this paper, the general open architecture of the DMS framework and the formal specification of the Data Definition Language (DDL) are introduced. A running example of multi-formalism model is used to illustrate the main concepts of the framework.

1 Introduction

The design of complex systems can be fruitfully supported by modeling: both qualitative and quantitative (e.g. performance and dependability) system properties can be analyzed on the models, and the results can be used to guide the design. Models are the basis of Model Driven Engineering (MDE) [1] techniques and the availability of flexible frameworks for models design and analysis is thus a relevant and timely issue. Such flexibility may require to represent the structure and the behaviour of the system by means of *multi-formalism* models and analyze them by means of *multi-solution* techniques. We talk about *multi-formalism* models when different modeling formalisms are used to represent in the most suitable way different aspects of the system; multi-formalism models usually require *multi-solution*, i.e. the (combined) use of different solvers to perform the analysis of the model.

It is thus very important to pursue the goal of embedding in a single tool the possibility of (1) building models by composition of sub-models (possibly reusing existing sub-models), and choosing among a set of different formalisms to express each sub-model; (2) defining and executing (more or less complex) solution procedures based on a set of solvers that can be used in isolation or in combination.

An example of a tool for performance and dependability analysis that goes in this direction is *Möbius* [2]: it includes several formalisms that can be integrated (even in a single model), and interpreted uniformly in the framework of a unique low level semantics: models analysis takes place at this common level by means of one of the many solvers included in *Möbius*. New formalisms can be embedded in the tool by defining (and implementing) the mapping toward the common semantic level. A few other tools, such as *Smart* [3] and *SHARPE* [4], allow the combined use of different formalisms, but the set of supported formalisms is predefined and closed.

The *Draw-Net Modelling System* (DMS) is a framework supporting the design and solution of models expressed in any graph-based formalism. The original idea behind the DMS, that differentiates it from the other approaches, is that it focuses on the integration of different existing tools to achieve the goal of solving multi-formalism models, rather than the creation of new tools. Moreover, the DMS is characterized by an open architecture such that the DMS can be customized for the design and the solution of models conforming to new graph based formalisms. This basic idea [5] has evolved, has been formalized and extended. Since its first version [5], the DMS framework has been designed to be very flexible and open to allow the inclusion of new formalisms without any programming effort (or at least very little programming effort): a user is free of integrating in the framework any (graph based) formalism and the corresponding solver. The integration of new formalisms in the framework is obtained by means of meta-modeling; a similar approach has been adopted in $ATOM^3$ [6].

The possibility to adapt the DMS model editor to design models of any graph based formalism, reminds the customization techniques of visual user interfaces in Model Driven Engineering (see e. g. [7]). However, the DMS has evolved in the context of modeling for performance and dependability evaluation, and as a consequence, DMS has some specific aspects usually not found in MDE tools.

With respect to our previous works, in this paper, we present the complete definition of the levels composing the current architecture of the DMS which was very briefly introduced in [8]. Here, we focus in particular on the *Data Definition Language* (*DDL*) which is the core of the DMS architecture: the DDL defines in an abstract way the elements for expressing formalisms and models.

The paper is organized as follows: in Sec. 2, the related literature is examined. Sec. 3 describes the architecture of the DMS and provides a formal description of the DDL. Sec. 4 describes an example of system that can be conveniently modeled and analyzed by following the DMS multi-formalism, multi-solution approach: this example supports the explanation of the concepts concerning the DDL. Sec. 5 shows the formalism definition process in the DMS; through the running example different situations of increasing complexity (from *simple* to *derived* and *composed* formalism) are presented. Sec. 5 describes also the *DNForGe* editor for the definition and the manipulation of

formalisms. Sec. 6 shows how models conforming to the formalisms defined in Sec. 5, can be built through the *Draw-Net tool*, the DMS model editor. Finally, Sec. 7 summarizes the ideas presented in the paper and defines some future work directions.

2 Related work

The DMS is a *language-oriented* framework for the definition of models and formalisms: thanks to its meta-modeling capabilities users can define their own modeling formalisms, and combine them as needed. This is its distinctive characteristic with respect to other similar projects.

Other tools allowing multi-formalism, like *Möbius* [9] or *Smart* [3], can be considered as *solution-oriented*: their main purpose is to propose a framework where efficient solution methods are implemented, based on some low level model. Multi-formalism is allowed by integrating formalisms that can be seen as high level counterparts of the basic low level model. Both the *syntax* and the *semantics* of these formalisms are defined. No meta-modeling facilities are provided to express the syntax: each time a new formalism must be included, a special purpose representation language, editor and parser must be defined and implemented; the semantics is then given by mapping the specific formalism into common low level concepts, so that the available solution algorithms can be applied to the new formalism through this mapping. Multi-solution is possible in both *Möbius* and *Smart*, in fact an overall solution can be obtained by allowing different models to exchange data, that is the value of a measure computed in one model can be assigned to an input parameter of another model. The introduction of new formalisms in the above tools requires some programming effort to adapt the new formalism to the common low level concepts managed by the implemented solvers. Moreover the extension of the set of available solvers or multi-solution strategies is an exclusive task of the tool development teams.

The *SHARPE* tool [4] allows to build and analyze models conforming to several formalisms and oriented to the performance or dependability evaluation, such as Fault Trees, Reliability Block Diagrams, Markov Chains, Stochastic Petri Net, etc. Moreover, *SHARPE* can deal with hierarchical models, i. e. models composed by several (heterogeneous) submodels at different levels: in a (sub)model, the property value of a modeling primitive can correspond to a measure computed on a submodel at lower level. For instance, the probability of a basic event in a Fault Tree can be set to be equal to the mean number of tokens in a place of a Stochastic Petri Net, or the probability of a state in a Markov Chain. Each submodel is analyzed with the proper technique (combinatorial analysis, state space analysis, etc.) and the results are passed from lower level submodels to the higher level ones. This tool was designed with a predefined set of modeling formalisms in mind, and it does not provide an easy way to include new formalisms or new solution modules.

The DMS differs from all the above mentioned tools since it focuses on the definition of languages to express models (i.e. on the *syntax*) and provides an easy way to include new formalisms; the choice of the most appropriate solvers and (multi)solution strategies are left to the user: the DMS only provides the mechanisms to connect existing solvers to its framework.

Another multi-solution framework is *MOSEL-2* [10]: it provides a high level language to specify models, that can then be translated in the specific modeling language of other tools (e.g. SPNP or TimeNET) for solving them. The DMS instead allows the user to choose the most effective representation: both high-level formalism (possibly domain specific ones) and low-level formalisms are allowed.

The idea of using meta-modeling to easily integrate new (graph based) formalisms can be found in $ATOM^3$ [6]: this framework allows to define also the semantics of new formalisms using *graph transformation grammars*. The DMS approach differs from that proposed in $ATOM^3$ since it does not provide any mechanism to define the semantics of the user-defined formalisms, but it leaves any semantic issue to the solvers. Graph transformation grammars are a very powerful and interesting way for translating a model from a given formalism into another one, and in the DMS framework they could play an important role in the definition of complex solution strategies. Another feature of the DMS that has no counterpart in $ATOM^3$ is the definition of a standard way to specify measures, and to define how to interchange them between editors and solvers. Measures are specific properties computable on a (sub)model.

Finally the DMS project closely integrates with the multi-formalism and multi-solution framework *OsMoSys* [11]. *OsMoSys* proposes a general and powerful approach to multi-solution, allowing very different solution techniques and tools to cooperate [12]: the core module allowing the integration of solvers is a *workflow engine*, through which models written using multiple formalism, can be solved using multiple solution components, orchestrated by a solution process (which can be user defined or automatically generated). The DMS framework provides the high-level user and solver interface to the *OsMoSys workflow engine*.

3 Architecture of the DMS

The DMS is a complete framework for the analysis of models that support both multi-formalism specifications and multi-solution analysis. The key aspect of the DMS is to be an *open framework*, where other components can be easily added to the system. The current DMS components are organized in a layered architecture as shown in Fig. 1.

The highest level identifies the user who can create or edit DDL based formalisms and models by means of the editors present in the *Editor level*; the

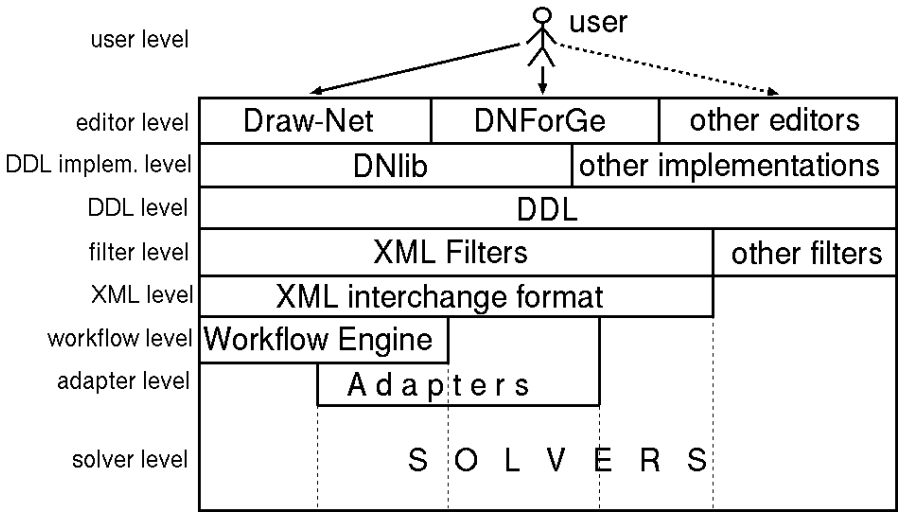

Fig. 1. The layered architecture of the DMS.

DMS provides two tools to this aim: *DNForGe* (Draw-Net FORmalism GEnerator) is the formalism editor, while *Draw-Net* is the model editor. The DDL is an abstract system of languages that allows the description of formalisms and models. In the *Editor level*, we can include other tools not provided by the DMS, but DDL based, for the manipulation of models and formalisms.

The *DDL level* concerns the abstract representation of the DDL described in Sec. 3.2, and is separated from the *Editor level* by the intermediate level called *DDL Implementation level* collecting the implementations of the DDL according to the several programming languages. Currently in the DMS, the DDL has been implemented only in Java language (*DNlib*); future implementations in other programming languages are possible and will be included in the *DDL implementation level*.

Below the *DDL level*, we find the *Filter level*; filters are DMS components for the conversion of formalisms and models expressed in DDL form, into a specific language adopted to save formalisms or models in files. Model solvers are software components that read the models, compute some sort of analysis and return some kind of result, such as performance or dependability measures; they are represented by the *Solver level*. Model solvers may have their own language to save models in a file, or they can exploit the XML interchange format defined inside the DMS. Such format allows to map in XML files, the data types composing formalisms and models expressed by the DDL. The XML formats defined inside the DMS will be considered in Sec. 3.5.

A model solver dealing with a model saved according to the XML interchange format of the DMS, may be able to directly parse the XML files describing the model; if not, an adapter has to be implemented in order to

map the XML files into other files still representing the model specification, but conforming to another model representation language.

An adapter consists of a program parsing a XML file and translating the model specifications in another language. XML parsing can be easily implemented by exploiting specific libraries which are available for the most common programming languages such as C, C++ and Java. The effort to translate the model from XML to a tool specific language, depends on the complexity of the model to be translated and on the complexity of the language used by the tool.

The *Adapter level* represents the set of the adapters and it connects the *XML level* with the set of the model solvers which require their own model representation language. The effort to produce adapters allows the DMS to be open to the integration of other solvers which exploit their own model representation language.

There are some solvers which can directly use the DMS XML interchange format: they are represented in Fig. 1 by a portion of the *Solver level* immediately below the *XML level*.

The *Workflow level* represents the workflow engines used to solve multi-formalism multi-solution; a workflow engine manages the solution of the submodels and of the global model; this is a complex task where several actions must be orchestrated; the workflow engine deals with aspects such as models decomposition, determining the order of solution of submodels, the invocation of filters or adapters, running solvers on submodels, passing results from a submodel to another, results storing and fetching, combining intermediate results to obtain the final result. The DMS supports hierarchical models definition and we assume that they are stored in XML files; so, solvers involved by a workflow engine may need adapters or not. For this reason, a portion of the *Adapter level* lies below the *Workflow level*. An example of workflow engine is that implemented in the OsMoSys framework [12].

3.1 Current state of implementation

The DMS architecture has not been completely implemented so far. The *DrawNet* tool for model editing is already available and is based on the DNlib. A prototype version of *DNForGe* has been developed before the definition of the DDL, so *DNForGe* is not yet based on the DDL, but on its own data structure to manage formalisms. The DDL has been fully implemented in Java language generating the DNlib. The XML filters towards the XML interchange format are available. As mentioned in the previous section, a workflow engine based on the DMS has been developed inside the OsMoSys framework.

Some solvers for models expressed in the XML interchange format have been developed in the past, but such format recently evolved, so those solvers need to be updated in order to deal with the current XML interchange format. These solvers allow to analyze *Fluid Stochastic Petri Nets* (FSPN) [13],

Extended Fault Trees [14] and *Dynamic Bayesian Networks* (DBN) [15]. Moreover, some work is in progress to support multi-solution in an automatic way.

3.2 The Data Definition Language

The DDL is the core of the DMS framework. It consists of a system of languages that allows the definition of a model at two levels: *the formalism level* and *the model level*.

The **formalism level** represents the languages used to describe models. It defines all the primitives that can be used to specify a model in a particular language. For example, it tells that a Petri Net is composed by Places, Transitions, and Arcs, and that a place contains tokens.

The **model level** contains the description of a system expressed in the corresponding formalism. It uses the primitives defined in the formalism to specify a model. For example it tells that a producer/consumer model described by a Petri Net is composed by two transitions (representing the producer and the consumer respectively) and a place (representing the buffer).

Sec. 3.3 and Sec. 3.4 give a formal description of formalisms and models inside the DDL, respectively.

3.3 Formalisms

A formalism is F defined as the tuple $F = \{E, P, C, S, H, T_P, L\}$ where E is the set of *Elements*; P is the set of *Properties*; C is the set of *Constraints*; $S : E \to 2^{(E \cup P)}$ is the structure function; $H : E \to 2^E$ is the inheritance function; $T_P : P \to \Sigma$ is the property typing function and $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is the set of allowed property types; L is the set of *Layers*.

Elements are the key feature of the DDL. Elements correspond to the entities that can be combined to construct a model, that is to represent any (sub)model expressed in a given formalism. Elements of a Petri Net formalism are for example Places, Transitions, Arcs and Petri Nets submodels.

Properties define the attributes associated with an element. For example a Petri Net's place has a "marking" property that counts the number of tokens contained in that place. Properties are typed: they can only contain values of a specific type. The typing function T_P assigns to each property its type. The set of property types Σ supported by the DDL, is divided into three classes: *atomic types*, *structured types* and *reference types*. Atomic types are the basic types and correspond to integers, real numbers, strings, Booleans and enumerations. Structured data types aggregate other data types to create more complex values. They are arrays, structures and unions (similar to the analogue concepts found in the *C* programming language). Reference data types contain reference to other elements of a model. Model references refer to entire models, element references are pointers to elements (possibly contained in other models), and property references are pointers to property values.

Element references may be used for example to define the starting and the ending point for an arc.

Each property type $\sigma \in \Sigma$ has its own set of possible values $\Gamma(\sigma)$. For example, for integer number $\Gamma(\mathsf{int}) = \mathbb{N}$, for real numbers $\Gamma(\mathsf{float}) = \mathbb{R}$, for an enumerated property ep ($T_P(ep) = \mathsf{enum}$) $\Gamma(\mathsf{enum}) = X = \{x_1, \dots x_n\}$ ($n \geq 2$), where X is the set of the values that can be assigned to the property ep. We will call Γ^* the set of all possible property values: $\Gamma^* = \{\forall \sigma \in \Sigma, \Gamma(\sigma)\}$.

Constraints are logical propositions that describe required consistency relations among elements and properties of a model. For example, in a Petri Net constraints tell that an arc can only connect places to transitions, and transitions to places, but not places to places or transitions to transitions.

Each element $e \in E$ may have some associated properties and a set of contained elements. This association is expressed by the *structure function* S. The ability of an element to be a container, and the fact that a formalism (a submodel) is an element itself, makes it possible to create multi-formalism models, by including in the model sub-models described in another formalism. Note that a formalism F is actually a container of different specification languages, which may contain several different paradigms. Each model will specify which of the available paradigms are actually used.

One of the key features of the DMS is the ability to define new elements by extending existing ones. One element can inherit properties and sub-elements from other elements. The inheritance is expressed by function H; the elements are partitioned into three subsets $E = E_a \cup E_p \cup E_c$: *abstract*, *private* and *concrete*. Abstract elements $e \in E_a$ cannot be instantiated directly in a model, but can be exploited to define a common set of properties and sub-elements that can be inherited by other elements $e' \in E_c$. Private elements $e \in E_p$ are excluded from inheritance. Concrete elements $e \in E_c$ can be both instanced and inherited.

The actual set of properties and sub-elements $\hat{S}(e)$ associated with an element $e \in E_c$ can be expressed as: $\hat{S}(e) = S(e) \cup \bigcup_{e' \in H(e)} \hat{S}(e')$. Inheritance is also used to define which elements are sub-formalisms, which are nodes and which are arcs in the graph that visually describes a model. Every formalism F, in order to be used in the DDL, must include three special abstract elements: $\{GraphBased, Node, Edge\} \subset E$. Every element that extends $GraphBased$ is a (sub)formalism. Every element that extends $Node$ is a node in the graph, and every element that extends $Edge$ is an edge.

Layers divides the elements and the properties of a formalism into classes. Each class contains elements and properties that refer to specific aspects of the formalism. The role of the layers is explained in Sec. 3.5

3.4 Models

A model is defined by the tuple $M = \{F, I, m_0, A, T, V, L\}$ where F is the formalism of the model; I are the element instances; $m_0 \in I$ is the main model; $A : I \to \{I \cup root\}$ is the model structure function; $T : I \to F.E_c \cup F.E_p$ is the

element typing function; $V : I \times F.P \rightarrow \{\Gamma^* \cup nil\}$ is the assignment function. Here, *nil* is a special element which represents the fact that a property has no assigned value. Finally, L is the set of the layers of the model; their meaning is explained in Sec. 3.5.

Every $i \in I$ represents an instance of an element of the formalism used in the model. $T(i)$ defines its type, that is the formalism element to which the instance corresponds. The element type must not be abstract. An instance i can contain other instances i'. This is specified by the model structure function $A(i') = i$. Note that since every element instance can be contained only in a single element, function A imposes a tree like structure to the model. If a node i'' is not contained in any other (it is a root), then $A(i'') = root$. Property values are specified by the assignment function V. In particular $V(i, p)$ represents the value of property $p \in F.P$ of the instance $i \in I$.

The instances enclosed in other instances, and the property assigned to a particular instance, must be compatible with the definitions and the constraints specified by the formalism. In other words:

$$\begin{aligned} \forall i, i' \in I : A(i') = i \Rightarrow T(i') \in F.\hat{S}(T(i)) \\ \forall p \in F.P, \forall i \in I : p \notin F.\hat{S}(T(i)) \Leftrightarrow V(i, p) = nil \\ \forall p \in F.P, \forall i \in I : V(i, p) \neq nil \Rightarrow V(i, p) \in \Gamma(F.T_P(p)) \\ \forall c \in F.C, \quad c = true \end{aligned}$$

The first proposition states that a model element may only contain element whose type is allowed by the formalism. The second states that all the properties associated with an element must be specified, and that an element cannot have other properties. The third one is used to state that properties can only assume values consistent with their type (as specified by the formalism). The last one is used to check the validity of the constraints.

A model M may actually include several models in its definition: this makes it possible to "reuse" a given parametric submodel (possibly extracted from a submodel library) by instantiating it several times in the global model definition, and be sure that an update in the parametric submodel will be automatically propagated to all its instances. However, a solver may require a starting point for the computation of its solution, m_0 represents the main model in I.

3.5 Formalism and model layers

In the DDL, formalisms and models are organized in layers; usually a formalism and the corresponding models have four standard different layers: the *structural layer*, the *query layer*, the *solution layer* and the *representation layer*.

The *Formalism Definition Layer* (FDL), the *Result Definition Layer* (RDL) and the *Formalism Representation Layer* (FRL) are related to a formalism.

The FDL is the structural layer and defines the elements of the formalism. For example in the Stochastic Petri Net formalism, the FDL contains the definition of places, transitions and arcs. The RDL is both the query layer and the solution layer of the formalism; this means that the RDL integrates both the definition of the possible queries and of the possible *results* available in a model conforming the formalism. For instance, in the Stochastic Petri Nets formalism, the RDL contains the transitions throughput, the mean number of tokens, the probability of reaching a specific marking and so on. The FRL is the representation layer and describes how the elements are graphically represented. Still in the case of Stochastic Petri Nets, the FRL contains the definition of the fact that places are drawn as circles and transitions as boxes.

The layers of a formalism are reflected in the models conforming to that formalism: the *Model Definition Layer* (MDL), the *Model Query Layer* (MQL), the *model ReSult Layer* (RSL) and the *Model Representation Layer* (MRL) are related to a model. The MDL contains the definition of the structure of the model. The MQL and RSL layers are used to interface with the solvers; the MQL contains *requests*, that is information that a solver can use to compute particular results. The RSL is not used by the modeler, but it is used by the solver to store the results it has computed. The MRL contains the graphical structure of the model indicating for instance the horizontal and the vertical position of each model composing element.

A concrete syntax for the DDL layers has been defined as basis of an XML interchange format for both formalisms, models and their layers. In order to simplify the handling of formalisms and models, the corresponding various layers are separated into different files.

For this reason, the DMS defines seven different XML based markup languages, one for each of the standard layers for both the formalisms and the models. Standard layers are summarized in Tab. 1.

Table 1. Standard layers for formalisms and models.

	Structure	Query	Results	Representation
Formalism	FDL	RDL	RDL	FRL
Model	MDL	MQL	RSL	MRL

4 Running example

Multi-formalism modeling means representing the system by means of several interacting sub-models, expressed with different formalisms.

In this section, we provide an example of a system that is conveniently represented by using multi-formalism, with the aim of evaluating the probability of the system to be failed at a certain time. Multi-formalism involves

multi-solution because each sub-model needs a specific solution method in order to be analyzed.

The system under study consists of an emergency lamp (L) which is initially off and is switched on by a controller (C) when the electric power (EP) is not supplied. We say that EP is *up* when EP is provided, and we say that EP is *down* when EP is missing. A sensor constantly indicates to the controller the presence or the absence of the electric power by sending to the controller a specific signal (S) corresponding to the state of EP: *up* or *down*.

The electric power is provided by two power suppliers: PS1 and PS2. PS2 is the spare component of PS1; more specifically, PS2 is a warm spare component. This means that PS1 is initially working, while PS2 is dormant, i. e. in a stand-by state. If PS1 fails, PS2 replaces PS1 in its function by turning from the dormant state to the working state. The time to fail of both PS1 and PS2 is ruled by a negative exponential distribution; the failure rate of PS1 is $\lambda = 1/8760 = 0.000114$, while the failure rate of PS2 changes according to its current state: while PS2 is dormant, its failure rate is $\alpha\lambda$, where $\alpha = 0.01$ is called the dormancy factor and reduces the probability of failure of PS2 with respect to the probability to fail of PS1. If PS2 is activated in order to replace PS1, the failure rate of PS2 becomes λ.

We assume that in the moment of activation of PS2, the controller C may fail with a certain probability (0.1). If C fails, the emergency lamp L will not be switched on in case of absence of electric power. Moreover, the sensor may send the wrong signal to C: the signal is *up* when EP is *down*, or the signal is *down* when EP is *up*, with a probability equal to 0.05. The probability of correct signal is instead 0.95. So, in case of electric blackout, L may not be switched on due to wrong signal S even though C is not failed.

The whole system fails when the electric power is missing and L has not been switched on. So, we are interested in computing this probability: $Pr\{L = off | EP = down\}$.

The multi-formalism model representing the behaviour of the system is composed by

- a *Generalized Stochastic Petri Net* (GSPN) [16] modeling the state variations of PS1 and PS2, together with the possible failure of C (Fig. 8);
- a *Bayesian Network* (BN) [17] modeling the signal S depending on the state of EP, and the possible activation of L depending on S and on the state of C (Fig. 7);
- a *Container* model indicating which results have to be computed and exchanged between the GSPN model and the BN model in order to obtain the measure of interest (Fig. 9).

In the next sections, we will refer to this example to explain how formalisms and models are expressed by the DDL. The example of multi-formalism use presented in the next sections is not trivial even if it is composed of a two level formalism hierarchy; in fact, to the best of our knowledge, in the literature there aren't examples of composition of these (basic) formalisms. This example

of multi-formalism model requires multi-solution to obtain some result: in this case the sequence of solvers application and results exchange is rather simple, and follows the container model structure, however in general much more complex solution schema might be required, and the corresponding solution process in that case should be explicitly defined [12]. In Sec. 5 we shall discuss a number of different ways of using multi-formalism.

5 Defining formalisms in the DDL

In the DDL, formalisms can be classified as *Simple formalisms* (Sec. 5.1), *Derived formalisms* (Sec. 5.2) and *Composed formalisms* (Sec. 5.3). The reference model (Sec. 6) is described using two formalisms (BN, GSPN) collected in a composed formalism (BN+GSPN). In particular, BN is a *Simple formalism* (Sec. 5.1) in the sense that it can be described in a single definition. GSPN is a derived formalism described by extending the Petri Net (PN) simple formalism and adding it GSPN specific modeling primitives.

5.1 Simple formalisms

A simple formalism defines the primitives of a graph based model whose elements can only be nodes and edges, with no sub-models. At the same time, a simple formalism does not inherit any element from a parent formalism (inter-formalism inheritance), except from *GraphBased*. However, in a simple formalism F_0, the intra-inheritance is possible; this means that an element $e \in F_0.E$ can inherit some of its properties from a set of parent elements $p_1, \ldots, p_m$ ($m \geq 1$) inside the same simple formalism; other properties can be defined specifically for the element e.

Let us consider the case of the BN simple formalism; Fig. 2 shows the elements of the BN formalism, using a UML-like graphic language where each box indicates the name of an element, its properties and the results computable on it. The use of UML is simply a way to graphically represent the elements of a formalism; actually, in the DMS, formalisms are represented by means of the DDL.

5.2 Derived formalisms

Derived formalisms are the result of the application of inter-formalism inheritance; this means that some of the elements of a derived formalism are inherited from another formalism (simple or derived); the other elements are formalism specific.

It is possible to define a formalism as *abstract*; in this way, it can only be used for derivation. Moreover, suppose that an element e inside a formalism F_1 is declared as *private*; if the formalism F_2 is derived from F_1, F_2 includes all the elements of F_1, except e.

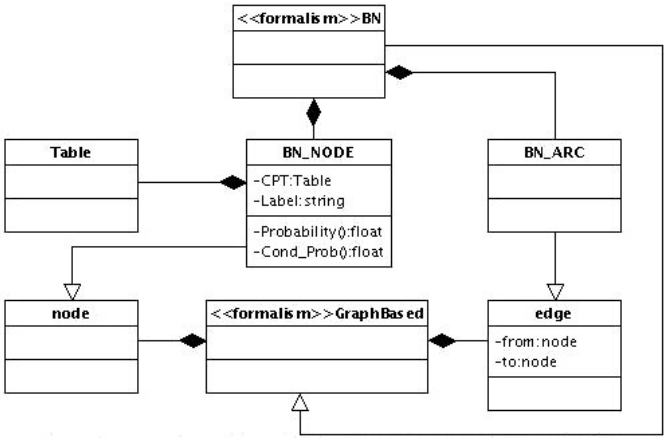

Fig. 2. UML-like diagram of the abstract representation of the BN formalism.

A case of derived formalism is GSPN derived from the PN simple formalism; Fig. 3 shows the elements of both the PN and GSPN formalism using an UML-like graphic language: GSPN inherits the same elements of PN, with the addition of some new elements: *INHIBITOR_ARC*, *TIMED_TRANSITION*, *IMMEDIATE_TRANSITION*. The last two elements are derived from the abstract element *GSPN_TRANSITION*, so the result *Throughput* is associated with both of them. No instances of *GSPN_TRANSITION* can be present in a model conforming the GSPN formalism.

5.3 Composed formalisms

In a multi-formalism model, we usually have a container model and a set of sub-models; the container model is an higher level model whose aim consists of containing several sub-models, and defining how each sub-model interacts with the others.

The elements to build a container model have to be defined in a *Composed formalism*. The main role of a composed formalism, is being the container of the several simple or derived formalisms, together with a set of elements (nodes, edges, measures) necessary to build the higher level models. Moreover, a composed formalism F_2 can derive from another composed formalism F_1; in such case, F_2 inherits all the elements of F_1 including nodes, edges and contained formalisms.

Fig. 4 shows the structure of an example of composed formalism called BN+GSPN and including the BN and the GSPN formalism. BN+GSPN is composed also by specific elements realizing the results exchange between submodels; such elements are $SOLVER$ which is a node, and two edges called $COMMUNICATION\_ARC$ and $SOLUTION\_ARC$. The property of the

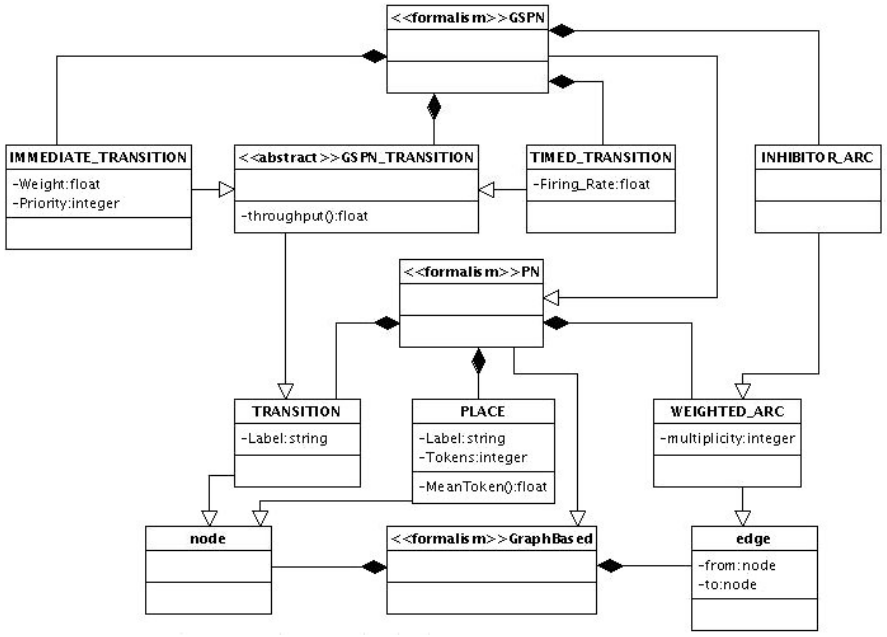

Fig. 3. UML-like diagram of the abstract representation of the GSPN formalism.

SOLVER node is *Solution_Tool* indicating the tool to be applied to a sub-model. The *SOLUTION_ARC* is used to connect *SOLVER* to a sub-model. *COMMUNICATION_ARC* is used to establish a connection between two sub-models with the consequent exchange of some values. A connection between two models establishes a sort of dependency of one model from the other. For instance, if a parameter of the model M_1 corresponds to a result to be computed on the model M_2, then M_1 depends on M_2. For this reason, a *COMMUNICATION_ARC* must have a verse pointing to the dependent sub-model; moreover, such an edge has some properties: *Result* in order to set which result has to be computed, *Object* to set the object of the result computation, *Variable* to indicate the name of the variable storing the result, once returned by the solution tool.

The formalisms included in this composed formalisms (Fig. 4) are BN and GSPN. In the BN+GSPN formalism, we define an element of type *measure*; a measure is a result concerning a (sub)model instead of a primitive element.

In [11] a distinction between explicit and implicit multi-formalism was made: the discussion presented in this paper concerns mainly explicit multi-formalism. Implicit multi-formalism can be used when some of the submodels composing the actual model to be used in the solution process are only implicitly represented by some "placeholder" in the (single formalism) model designed by the user: the placeholder is then automatically transformed into

the final submodel (and composed with the complete model) at solution time; an example of implicit multi-formalism can be found in [14]. Multi-formalism models usually require multi-solution: in the simplest cases the solution process can be directly inferred from the model structure (as in the example presented in this paper) however in general the solution process can be more complex and needs to be explicitly described: an example of language for expressing solution processes and of general multi-solution tool has been presented in [12].

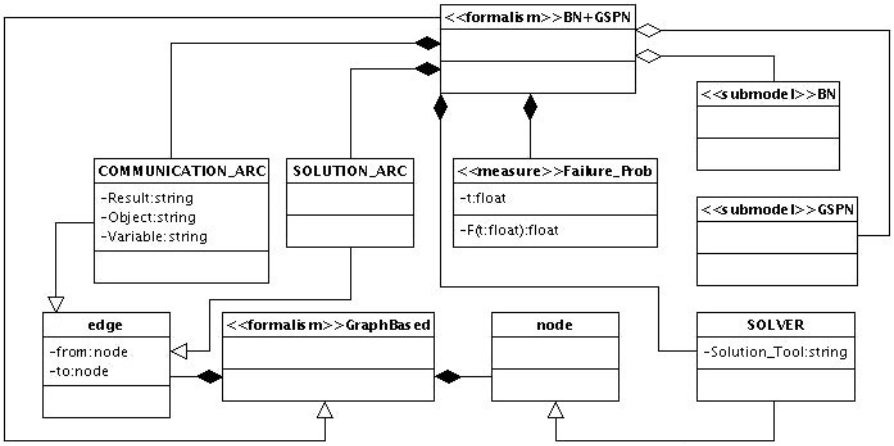

Fig. 4. UML-like diagram of the abstract representation of BN+GSPN.

5.4 *DNForGe*

The *Draw-Net tool* allows to create or edit any graph based model whose formalism has been previously defined. Due to the complexity of a formalism specification, and the high number of parameters to define, building a formalism manually by writing directly its XML code, would be unpractical; so a way to simplify the specification of a formalism, became necessary.

For this reason, a graphical interface called *DNForGe* has been developed with the aim of creating and editing formalisms for the DMS. By means of *DNForGe*, the user can manipulate the definition of a formalism avoiding to deal with the XML code. Such code is automatically generated or updated by *DNForGe*.

The main window of *DNForGe* (Fig. 5) displays in a tree graphical structure the hierarchy of the formalisms collected in a composed formalism; by means of the main window, the user can modify the formalisms hierarchy by adding or removing formalisms inside composed formalisms. Moreover, from

this window, the user can select a single formalism and edit it in a specific window. (Fig. 6). In this window the user can add or remove formalism elements; he can also select a certain element and edit its properties by means of a further window.

In general, *DNForGe* allows the user to define formalisms of any kind (simple, derived, composed), exploiting all the aspects described in the previous sections, such as intra-formalism and inter-formalisms inheritance, abstract and private elements.

Let us consider the GSPN formalism; it has been built using *DNForGe*. The content of the XML file storing the FDL layer of such formalism, follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE fdl SYSTEM '../../dtd/fdl.dtd'>
<fdl main="GSPN">
  <include src="base/GraphBased.fdl" />
  <include src="base/Instantiable.fdl" />
  <parent ref="PN" />
  <elementType name="GSPN_TRANSITION" type="abstract">
    <parent ref="TRANSITION" />
  </elementType>
  <elementType name="IMMEDIATE_TRANSITION" >
    <parent ref="GSPN_TRANSITION" />
    <propertyType name="Weight" type="float" default="1.0" />
    <propertyType name="Priority" type="integer" default="1" />
  </elementType>
  <elementType name="TIMED_TRANSITION" >
    <parent ref="GSPN_TRANSITION" />
    <propertyType name="Firing_Rate" type="float" default="1.0" />
  </elementType>
  <elementType name="INHIBITOR_ARC" >
    <parent ref="WEIGHTED_ARC" />
  </elementType>
</fdl>
```

The description of the RDL layer of the GSPN formalism is contained in this XML file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE rdl SYSTEM "../../dtd/rdl.dtd">
<rdl main="GSPN">
  <parent ref="PN" />
  <elementType name="GSPN_TRANSITION">
    <resultType name="Throughput"/>
  </elementType>
  <elementType name="IMMEDIATE_TRANSITION">
    <parent ref="GSPN_TRANSITION" />
```

```
  </elementType>
  <elementType name="TIMED_TRANSITION">
    <parent ref="GSPN_TRANSITION" />
  </elementType>
  <elementType name="INHIBITOR_ARC">
    <parent ref="WEIGHTED_ARC" />
  </elementType>
</rdl>
```

We avoid to report the XML representation of the FRL layer of the formalism.

Fig. 5. Screenshot of the "*DNForGe: Composer*" window.

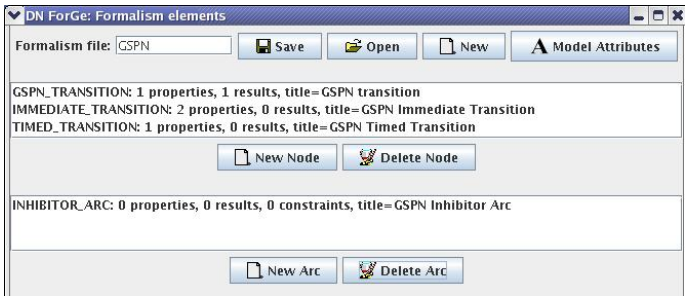

Fig. 6. Screenshot of the "*DNForGe: Formalism Elements*" window.

6 Building models

In this section the multi-formalism model representing the behaviour of the system proposed in Sec. 4, is described; the model is composed by a container

model (Fig. 9) conforming to the BN+GSPN formalism, and two submodels: BN_0 (Fig. 7), and $GSPN_1$ (Fig. 8) conforming to the BN and GSPN formalism respectively (Sec. 5). Each submodel represents a specific feature of the system behaviour, as mentioned at the end of Sec. 4. The complete multi-formalism model is drawn by means of the *Draw-Net tool* (Fig. 8). We avoid the description of the submodels BN_0 and $GSPN_1$, and we concentrate our attention on the container model (Fig. 9). According to the formal definition of model given in Sec. 3.4, the container model is the main model (m_0).

Some examples of the XML representation of models are reported in [18].

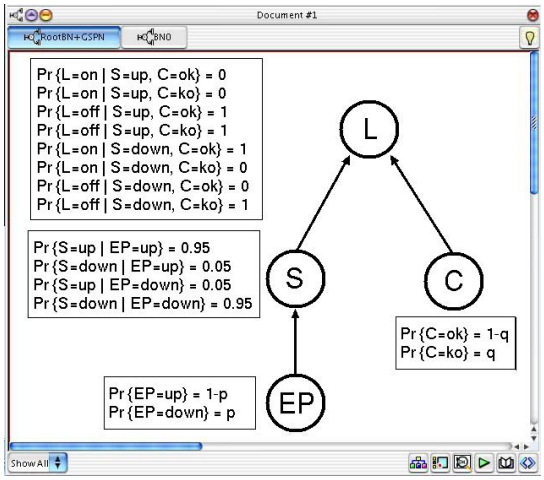

Fig. 7. Submodel BN_0.

Container model

Fig. 9 shows the container model conforming the BN+GSPN formalism (Sec. 5.3) and setting the interconnections among the submodels BN_0 and $GSPN_1$. A measure named *Failure_Prob* concerns the whole multi-formalism model and indicates the probability of the system to be failed at a certain time.

Several instances of $COMMUNICATION\_ARC$ connecting submodels, indicate the exchange of results among submodels: in BN_0 (Fig. 7), the entries of the Conditional Probability Table (CPT) of the node EP contain the variable p, while the entries of the CPT of the node C contain the variable q. The properties of the arcs connecting $GSPN_1$ to BN_0 indicate that p and q must be computed on $GSPN_1$ as the mean number of tokens present in the places called EP_down and C_ko respectively. Since the number of tokens inside such places can be 0 or 1, their mean number of tokens at time t will be the probability of these places to be marked at time t.

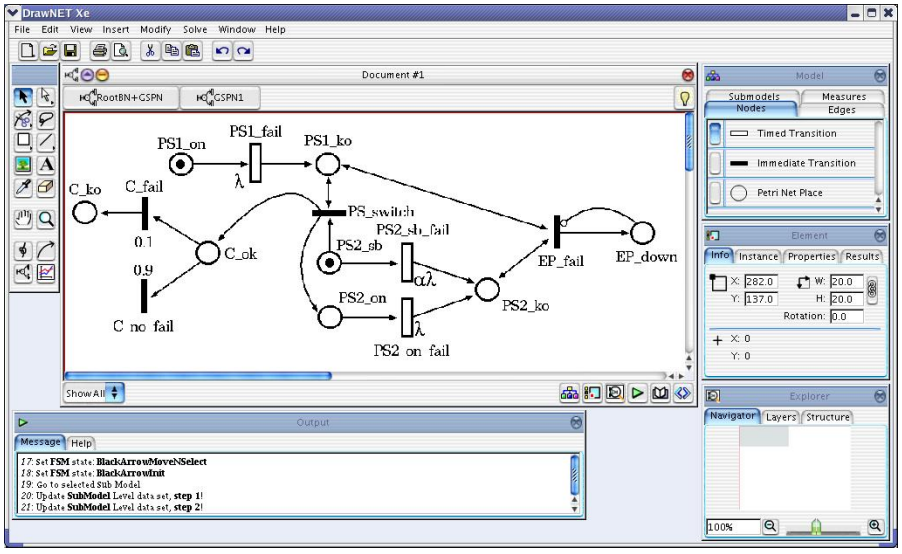

Fig. 8. Screenshot of the *Draw-Net tool* showing the submodel $GSPN_1$.

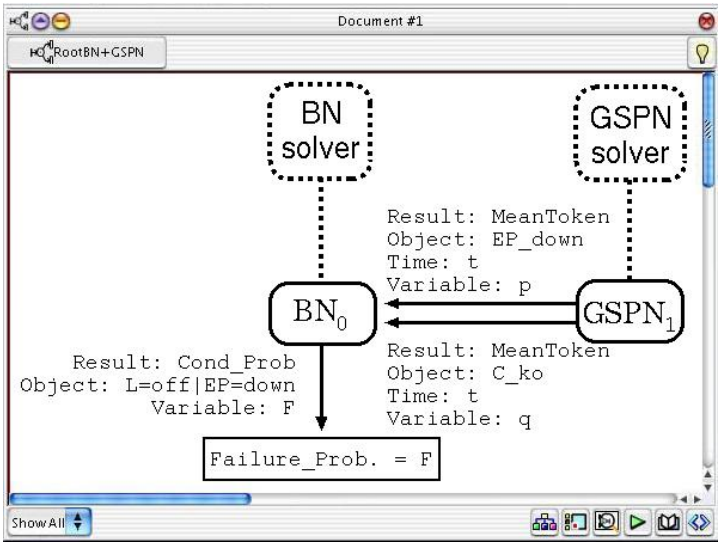

Fig. 9. Container model.

In the container model, the solver to be used for each submodel, is indicated by means of instances of $SOLVER$ and $SOLUTION\_ARC$.

Finally, BN_0 is connected to the measure *Failure_Prob* relative to the whole multi-formalism model, with the specification that such measure is equal to the conditional probability $Pr\{L = off|EP = down\}$ to be computed on BN_0. We suppose that a workflow engine (WE) is available to deal with models conforming the BN+GSPN composed formalism; such workflow engine can determine the order of solution of the submodels given the way they are connected in the container model (Fig. 9). Moreover, we suppose that the node *BN solver* (instance of $SOLVER$) in the container model, indicates the tool *DBNet* [15] as BN analyzer, while the node *GSPN solver* indicates that *GreatSPN* [19] is the tool to analyze GSPNs. According to these assumptions and assuming that the modeler has set t to 1000 h in the container model, the computation of the probability of the system failure at time t, will follow these steps:
1. the tool *GreatSPN* performs the analysis of the submodel $GSPN_1$ returning these measures: $p = E(\#EP\_down, t)$, $q = E(\#C\_ko, t)$, i. e. p is the mean number of tokens inside the place EP_down at time t, and q is the mean number of tokens inside the place EP_down at time t;
2. the WE passes the values of the variables p and q to the submodel BN_0 where p and q are present in some CPTs;
3. the tool *DBNet* performs the analysis of BN_0 returning the conditional probability $F = Pr\{L = off|EP = down\}$;
4. in the container model, the WE sets the value of the measure *Failure_Prob* at time t, to the value of F.

Currently, this process is executed in a semi-automatic way.

7 Conclusions and future work

This paper has presented the multi-formalism multi-solution DMS framework, focusing on the central role played by the DDL in the open architecture of the DMS.

The DDL offers a meta-modeling feature that allows the creation of new formalisms: intra-inheritance and inter-inheritance allow to derive new formalism by extending existing ones, and to integrate different formalisms in a composed formalism. This last possibility is the basis for multi-formalism modeling.

According to the DDL, models conforming to a certain formalism can be constructed by instantiating formalism elements; as mentioned above, multi-formalism models can be constructed by instantiating (sub)models expressed in different formalisms within a container model conforming to a composed formalism.

The current DMS implementation includes a library (DNlib) implementing the DDL, and two editors, *DNForGe* and *Draw-Net*, for the manipulation of formalisms and models respectively.

The DMS is an open and extensible framework: other DDL-based editors may be implemented and added to the DMS, exploiting the DNlib, and solvers can be connected to the framework by exchanging the XML interchange files representing the different layers of a DMS model, or by exchanging solver specific files (in this last case appropriate filters or adapters must be constructed).

Future work on the DMS will be mainly devoted to interfacing the DMS with several solvers: the first goal consists in building the connection between the DMS and a number of miscellaneous solvers exploiting the previous version of the XML interchange format (see Sec. 3.1).

The second goal is to interface the DMS framework with other solvers, developed within other projects: among the others it is our intention to interface the GSPN and SWN (Stochastic Well-formed Nets) [20] solvers developed within *GreatSPN* and related projects [21].

ACKNOWLEDGEMENTS

This work was supported by the Italian Ministry of Education, University and Research (MIUR) in the framework of the FIRB-Perf project.

References

1. Douglas C. Schmidt. Model driven engineering, *guest editor's introduction. IEEE Computer, Special Issue on Model Driven Engineering*, pages 25–31, Feb. 2006.
2. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. Doyle, W. Sanders, and P. G. Webster. The Möbius Framework and its Implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.
3. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Smart: Stochastic model analyzer for reliability and timing. In *Tools of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 29–34, Aachen, Germany, Sept. 2001.
4. R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems; An Example-based Approach Using the SHARPE Software Package*. Kluwer Academic Publisher, 1996.
5. G. Franceschinis, M. Gribaudo, M. Iacono, V. Vittorini, and C. Bertoncello. DrawNet++: a flexible framework for building dependability models. In *In Proc. of the Int. Conf. on Dependable Systems and Networks*, Washington DC, USA, June 2002.
6. J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-Modeling and Graph Grammars for Multi-Paradigm Modeling in $ATOM^3$. *Journal of Software and System Modeling*, 3(3):194–209, August 2004.
7. I. Bull and J. M. Favre. Visualization in the Context of Model Driven Engineering. In *Proceedings of the International Workshop on Model Driven Development of Advanced User Interfaces*, Montego Bay, Jamaica, October 2005.

8. M. Gribaudo, D. Codetta-Raiteri, and G. Franceschinis. Draw-Net, a customizable multi-formalism multi-solution tool for the quantitative evaluation of systems. In *Proceedings of the 2nd International Conference on Quantitative Evaluation of Systems*, pages 257–258, Turin, Italy, September 2005.
9. J. M. Doyle. Abstract models specification using the Möbius modeling tool. Master's thesis, University of Illinois, 2000.
10. Patrick Wuechner, Hermann de Meer, Joerg Barner, and Gunter Bolch. MOSEL-2 – A Compact But Versatile Model Description Language And Its Evaluation Environment. In *Proc. of MMBnet'05, Hamburg*, pages 51–59, 2005.
11. V.Vittorini, M. Iacono, N. Mazzocca, and G. Franceschinis. The OsMoSys approach to multi-formalism modeling of systems. *Journal of Software and System Modeling*, 3(1), March 2004.
12. M. Gribaudo, N. Mazzocca, F. Moscato, and V. Vittorini. Multisolution of Complex Performability Models in the OsMoSys/DrawNet Framework. In *Proc. 2nd Int. Conf. on the Quantitative Evaluation of Systems*, pages 85–94, Torino, Italy, Sept. 2005.
13. M. Gribaudo. FSPNEdit: A fluid stochastic Petri net modeling and analysis tool. Technical report, Tools of Aachen 2001 - International Multiconfernce on Measurements Modelling and Evaluation of computer Communication Systems - University of Dortmund, Bericht No. 760/2001, 2001.
14. D. Codetta-Raiteri. *Extended Fault Trees Analysis supported by Stochastic Petri Nets.* PhD thesis, Dipartimento di Informatica, Università di Torino, November 2005. http://www.di.unito.it/∼phd/phd_theses.html.
15. S. Montani, L. Portinale, A. Bobbio, M. Varesio, and D. Codetta-Raiteri. A tool for automatically translating Dynamic Fault Trees into Dynamic Bayesian Networks. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 434–441, Newport Beach, CA USA, January 2006.
16. M. Ajmone-Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* J. Wiley and Sons, 1995.
17. L. Portinale and A. Bobbio. Bayesian networks for dependability analysis: an application to digital control reliability. In *Proc. 15th Conference on Uncertainty in Artificial Intelligence (UAI 99)*, pages 551–558, July 1999.
18. M. Gribaudo, D. Codetta-Raiteri, and G. Franceschinis. The Draw-Net Modeling System: a framework for the design and the solution of single-formalism and multi-formalism models. Technical Report TR-INF-2006-01-01-UNIPMN, Dipartimento di Informatica, Università del Piemonte Orientale, January 2006.
19. G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation, special issue on Performance Modeling Tools*, 24(1&2):47–68, November 1995.
20. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications. *IEEE Transactions on Computers*, 42:1343–1360, 1993.
21. J-M. Ilié, S. Baarir, M. Beccuti, S. Donatelli, C. Dutheillet, G. Franceschinis, R. Gaeta, and P. Moreaux. Extended SWN solvers in GreatSPN. In *Proc. of the 1st Int. Conference on Quantitative Evaluation of Systems (QEST'04)*, pages 324–325, Twente, The Netherlands, September 2004. IEEE Computer Society.