Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Via Bellini 25/G, 15100 Alessandria
http://www.di.unipmn.it

The *Draw-Net Modeling System*: a framework for the design and the solution of single-formalism and multi-formalism models

Authors: *Marco Gribaudo (marco.gribaudo@di.unito.it),*
Daniele Codetta-Raiteri (daniele.codetta_raiteri@unipmn.it),
Giuliana Franceschinis (giuliana.franceschinis@unipmn.it).

TECHNICAL REPORT TR-INF-2006-01-01-UNIPMN
(January 2006)

The University of Piemonte Orientale Department of Computer Science Research Technical Reports are available via WWW at URL `http://www.di.mfn.unipmn.it/`.

Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2005-06 *Compressing and Searching XML Data Via Two Zips*, Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S., December 2005.

2005-05 *Policy Based Anonymous Channel*, Egidi, L., Porcelli, G., November 2005.

2005-04 *An Audio-Video Summarization Scheme Based on Audio and Video Analysis*, Furini, M., Ghini, V., October 2005.

2005-03 *Achieving Self-Healing in Autonomic Software Systems: a case-based reasoning approach*, Anglano, C., Montani, S., October 2005.

2005-02 *DBNet, a tool to convert Dynamic Fault Trees to Dynamic Bayesian Networks*, Montani, S., Portinale, L., Bobbio, A., Varesio, M., Codetta-Raiteri, D., August 2005.

2005-01 *Bayesan Networks in Reliability*, Langseth, H., Portinale, L., April 2005.

2004-08 *Modelling a Secure Agent with Team Automata*, Egidi, L., Petrocchi, M., July 2004.

2004-07 *Making CORBA fault-tolerant*, Codetta Raiteri D., April 2004.

2004-06 *Orthogonal operators for user-defined symbolic periodicities*, Egidi, L., Terenziani, P., April 2004.

2004-05 *RHENE: A Case Retrieval System for Hemodialysis Cases with Dynamically Monitored Parameters*, Montani, S., Portinale, L., Bellazzi, R., Leonardi, G., March 2004.

2004-04 *Dynamic Bayesian Networks for Modeling Advanced Fault Tree Features in Dependability Analysis*, Montani, S., Portinale, L., Bobbio, A., March 2004.

2004-03 *Two space saving tricks for linear time LCP computation*, Manzini, G., February 2004.

2004-01 *Grid Scheduling and Economic Models*, Canonico, M., January 2004.

2003-08 *Multi-modal Diagnosis Combining Case-Based and Model Based Reasoning: a Formal and Experimental Analisys*, Portinale, L., Torasso, P., Magro, D., December 2003.

2003-07 *Fault Tolerance in Grid Environment*, Canonico, M., December 2003.

2003-06 *Development of a Dynamic Fault Tree Solver based on Coloured Petri Nets and graphically interfaced with DrawNET*, Codetta Raiteri, D., October 2003.

Contents

The *Draw-Net Modeling System*: a framework for the design and the solution of single-formalism and multi-formalism models

Marco Gribaudo
Dipartimento di Informatica
Università di Torino
Corso Svizzera, 185
10149 Torino, Italy
marcog@di.unito.it

Daniele Codetta-Raiteri*, Giuliana Franceschinis
Dipartimento di Informatica
Università del Piemonte Orientale
Via Bellini, 25/G
15100 Alessandria, Italy
{raiteri, giuliana}@mfn.unipmn.it

Abstract

This paper presents the *Draw-Net Modeling System*, a framework for the design and 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. The proposed approach allows to represent different aspects of a system with the most suitable formalism, and to analyze different properties by means of the most appropriate solution algorithm and software tool. After having introduced the XML based language family that can be used to define new formalisms (possibly extending existing ones), and to describe models, the main functions implemented in the framework are presented, and the general (open) architecture of the framework is discussed. Some examples of formalisms are presented, showing the possible applications of the concepts on which the *Draw-Net Modeling System* framework is based.

1 Introduction

The performance and dependability evaluation of complex systems may require to represent the behaviour of the system by means of *multi-formalism* and *multi-solution* models. 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 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

*Corresponding author: daniele.codetta_raiteri@unipmn.it

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 that goes in this direction is *Möbius* [1, 2, 3, 4]: it includes several formalisms that can be integrated (even in a single model), and interpreted uniformly in the framework of a single low level semantic: 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* [5] and *Sharpe* [6, 7], allow the combined use of different formalisms, but usually the set of supported formalisms is predefined and closed.

This paper describes the *Draw-Net Modeling System* (DMS) [8], a framework supporting the design and solution of models expressed in any graph-based formalism. The system is characterized by an open architecture and includes an XML based language family that can be used to define existing as well as new formalisms and multi-formalism models expressed through such formalisms.

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 can be customized for the design and the solution of models conforming to new graph based formalisms.

Since its first version [9, 10], 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. This basic idea [9] has evolved, has been formalized and extended.

The paper is organized as follows: Sec. 2 describes the basic components of the DMS architecture and in particular the *Data Definition Language* (DDL) that defines in an abstract way the elements for expressing formalisms and models; the set of XML based languages used by the tool for the exchange of formalisms and models is also introduced in this section. Sec. 3 describes a system that can be conveniently modeled and analyzed by following the DMS multi-formalism, multi-solution approach: it is used as a running example to illustrate in an intuitive way all the relevant concepts throughout the paper. Sec. 4 shows the formalism definition process in the DMS, presenting through the running example different situations of increasing complexity (from *simple* to *derived* and *composed* formalism), and describing the *DNForGe* editor, one of the DMS components. Sec. 5 shows how models can be built through the *Draw-Net tool* (another editor composing the DMS): a brief discussion of the DMS software architecture highlighting its extensibility is also

included. Sec. 6 presents a core set of solvers currently included in the DMS. Finally, Sec. 7 summarizes the ideas presented in the paper and defines some future work directions.

2 *Draw-Net Modeling System* architecture

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.

Components can be divided into two categories: *Editors* and *Solvers*. Editors are software components that allow the user to define formalisms and models. Solvers are software components that read the models written by the editors, compute some sort of analysis, and complete the models definitions by the results they have computed.

The DMS comprises other components that can be used at different extents to obtain the desired features. In particular, the DMS is composed by:

- A library to access the features of the DMS framework (called *DNlib*).
- A system of languages called DDL for the definitions of formalisms and models (Sec. 2.1).
- A graphical user interface for the definition of formalisms called *DNForGe* (Sec. 4.4).
- A graphical user interface for the definition of models called the *Draw-Net tool* (Sec. 5.1).
- A set of solution components (called the *Native Solvers*) that can compute performance (or dependability) indices of models specified using one of the formalisms belonging to the "native-formalism" archive (Sec. 6).
- A set of *filters* for the translation of models expressed in DDL, into the language adopted by a solver.

Currently, the framework provides two graphical user interfaces (*DNForGe* and the *Draw-Net tool*) and a set of solution components (the *Native Solvers*), but users can add other editors and other solvers to the system by exploiting the DNlib.

Editors and solvers exchange models and formalisms described with a specific system of languages based on XML: the *Data Definition Language* (DDL). Any editor capable of describing models and formalisms using the DDL can take advantage of the solutions provided by the solvers. In the same way, solvers that can compute results on models and formalisms described using the DDL, can provide their service to all the editors. In order to simplify the creation of solvers and editors, the framework provides a library, called the DNlib, that implements the DDL.

The DDL is an abstract system of languages that allows the description of models and formalisms. The solution of a model expressed by means of the DDL, may require the translation of the model into the language adopted by the solver. This can be carried out in many different ways. In order to achieve the maximum flexibility, the translation is done by specialized classes called *filters*. The DMS proposes a set of standard XML-based formats and filters. These formats will be considered in Sec. 2.4. User defined formats can be added by including new filters: this allows the direct interaction with existing solvers or editors without any intermediate translation steps.

2.1 The Data Definition Language

The DDL is the core of the DMS framework. It consist of a system of languages that allows the definition of a performance 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 in the corresponding formalism. It uses the primitives defined in the corresponding formalism to specify a particular 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 where the produced parts are waiting to be consumed).

Sec. 2.2 and Sec. 2.3 explains the meaning of a formalism and of a model inside the DDL, respectively.

2.2 Formalisms

A formalism is defined as

$$F = \{L, E, P, C, e_0, S, H\}$$

where

- L is the set of *Layers*;
- E is the set of *Elements*;
- P is the set of *Properties*;
- C is the set of *Constraints*;
- $e_0 \in E$ is the main formalism;
- $S : E \rightarrow 2^{(E \cup P)}$ is the structure function;

- $H : E \to 2^E$ is the inheritance functions.

Elements are the key feature of the DDL. Elements correspond both to formalisms primitives and to entire formalisms. Elements for a Petri Net are for example Places, Transitions, Arcs and the Petri Nets themselves. *Properties* define the attributes associated with an element. For example a Petri Net's place has a property that counts the number of tokens contained in that place. Properties are typed: they can only contain values of a specific type. A special property type is the *Element reference* type. It allows to store the reference to other elements, and it is used for example to define the starting and the ending point for an arc. Properties are partitioned into two sets $P = P_I \cup P_O$: the *input properties* P_I that are specified by the editors and contains data belonging to the definition of the model, and *output properties* P_O (also referred as *results*) that are filled in by the solvers and returned to the editors as the results of their computations. Input properties may have associated a default value.

Constraints are logical propositions that tell whether some particular relations between elements and properties are possible or not. 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 $e \in E$ has associated some properties and a set of elements that it can contain. This relations is expressed by the *structure function* S. The ability of an element to be a container, and the fact that a formalism is an element itself, make it possible to create multi-formalism models, by including sub-models described in another formalism. However this may also be a source of confusion, since the formalism structure F may be confused with a specific formalism. In multi-formalism models, F may contain more than one single formalism.

In order to define a single starting point (that is a container formalism that can be used to define sub-models in specific sub-formalisms), an element $e_0 \in E$, called the main formalism, is used to represent the outer-most element of E. Since there is a single main formalism in F, when we will refer to a formalism F, we will be actually referring to e_0.

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 form other elements. The inheritance is expressed by the function H. Due to inheritance, the elements are partitioned into two subsets $E = E_a \cup E_c$: *abstract* and *concrete*. Abstract elements $e \in E_a$ cannot be instantiated directly in model, but can be exploited to define a common set of properties and sub-elements that can be reused by other elements $e' \in E_c$. The actual set of properties and sub-elements $\hat{S}(e)$ associated to 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. Usually a model has three different layers: the *structural layer*, the *solution layer* and the *representation layer*. The structural layer contains the definition of the features that characterize the structure of a model. For example in a Petri Net, it contains places, transitions, tokens, transition rates, arcs, arcs weights an so on. The solution layer contains the parameters computed by a solver. In the Petri Net example, it may contain transition throughputs, mean number of tokens, probability of reaching a specific marking and so on. The representation layer contains the definition of all the graphical aspects of a model, such as the fact that places are drawn by circles and transitions by boxes.

2.3 Models

A model is defined by its reference formalism, as a set of instances:

$$M = \{F, I, m_0, A, T, V\}$$

where

- F is the formalism of the model;
- I are the element instances of $F.E$;
- $m_0 \in I$ is the main model;
- $A : I \to 2^I$ is the inclusion function;
- $T : I \to F.E_c$ is the element typing function;
- $V : I \times F.P \to \{\Sigma \cup \{nil\}\}$ is the assignment function. Σ represents the set of all the possible values that a property can hold (i.e. integer, floating points numbers, strings, booleans, element references), and nil represents the fact that a property is not associated with a particular element.

Every $i \in I$ represents an instance of a primitive 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 $i' \in A(i)$, as specified by the inclusion function A. 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 $i \in I$.

Table 1: XML based interchange formats

	Formalism	Model
Structure	FDL	MDL
Results	RDL	MQL
Representation	FRL	MRL

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:

$$T(i) \in F.E_c \wedge \forall i' \in A(i) \Rightarrow$$

$$\Rightarrow T(i') \in F.\hat{S}(T(i)) \wedge \forall p \in F.P, p \notin F.\hat{S}(T(i)) \Rightarrow$$

$$\Rightarrow V(i,p) = nil \wedge c = true, \forall c \in F.C$$

Also a model M may contain more than a single model. This enables the support for model classes [11]. m_0 represents the main model in $M.I$, and it must be compatible with the main formalism, that is: $T(m_0) = F.e_0$. In the following, we will address m_0 simply with the term model. All this aspects will be clarified in Sec. 5.

2.4 XML format

A set of standard filters to represent the data types expressed by the DDL is defined in the DMS. This set of filters uses the XML interchange format. In particular it defines six different XML based markup languages: one for each of the standard layers that compose both the formalisms and the models. Table 1 summarizes the languages composing the XML interchange format.

The *Formalism Definition Layer* (FDL), the *Result Definition Layer* (RDL) and the *Formalism Representation Layer* (FRL) are related to the formalism; they define its primitives, which results may be computed, and how the elements are graphically represented, respectively. The *Model Definition Layer* (MDL), the *Model Query Layer* (MQL) and the *Model Representation Layer* (MRL) are related to a model; they contain the definition of the model, which results need to be computed, and the graphical structure of the model, respectively. The *FDL* and *MDL* of the example described in Sec. 3 are presented in the Appendix.

3 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 system unavailability, i. e. the probability that the system is not working at a certain time. Multi-formalism involves multi-solution because each sub-model needs a specific solution method in order to be analyzed.

Such system consists of a lighting plant composed by a set of 9 lamps ($L1, L2, \ldots, L9$), one electric power supplier (S), one battery (B) and a controller (C).

Initially, all the lamps are working and are electrically supplied by S; each lamp may fail after a random period of time which is a random variable ruled by a negative exponential distribution whose failure rate is $\lambda_L = 1/4320h = 0.000231481h^{-1}$; also the electric power supplier S may fail, but it can be repaired. The time to fail and the time to repair of S are random variables obeying the negative exponential distribution: the failure rate of S is $\lambda_S = 1/8760h = 0.000114155h^{-1}$; the repair rate of S is $\mu_S = 1/24h = 0.0416667h^{-1}$.

While S is under repair, it is replaced in its function by the battery B; the charge level of B changes during the time, it is expressed as a percentage and it is indicated by l_B. Such value is initially equal to 100%; while B replaces S, l_B gradually decreases: in order to exhaust its charge, the battery B needs to supply electric power for $168h$; when $l_B = 0\%$ the battery can not supply the lamps.

The aim of the controller C is switching the electric power supply from S to B when S fails; the controller is a multi-state component: it is initially working, but it may turn to stuck. From the stuck state, the controller can turn back to the working state, or it can turn to the definitive failed state. Only in the working state, the controller can fulfil its aim. The duration of the controller states are random variables which will obey to the negative exponential distribution; Fig. 8 shows the possible state transitions of C with the corresponding rates.

The system works correctly if at least 5 of the 9 lamps are not yet failed, and the electric power is supplied by S or B. The system fails if 5 lamps fail, or the electric power is not supplied by neither S nor B.

Three sub-models are involved in order to represent the whole system behaviour:

- a *Parametric Fault Tree* (PFT) [12, 13, 14] (Fig. 6) to represent the combination of component failure events leading to the system failure;
- a *Fluid Stochastic Petri Net* (FSPN) [15, 16, 17] (Fig. 7) to represent the failure of S and its replacement by B during the repair, with the consequent wear of B during the repair time;
- a *Continuous Time Markov Chain* (CTMC) [7, 18] (Fig. 8) to represent the controller state transitions.

In the next sections, we will refer to such models to explain how formalisms and models are built in *DNForGe* and *Draw-Net tool* respectively.

4 Defining formalisms in the DMS

The reference model (Sec. 5) is described using three different formalisms (PFT, FSPN, CTMC) collected in a composed formalism. In particular, CTMC and PFT are *Simple formalisms* (Sec. 4.1) in the sense that they can be described in a single definition. FSPNs can be more easily described by extending the Generalized Stochastic Petri Nets (GSPN) [19] and adding them the fluid features. We call these kind of formalisms *Derived formalisms* (Sec. 4.2). The *Composed formalisms* (Sec. 4.3) can be described as containers of other previously defined formalisms, with the addition of special composition primitives.

In Sec. 4.4, we will show the graphical user interface called *DNForGe* implemented to manipulate such formalisms.

4.1 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 $e \in F_0.E$ ($e \neq e_0$) 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.

In a simple formalism, it is also possible to define elements as *abstract*; such elements can not be instantiated in the model; their unique aim is being the parent of other elements. This is useful when several elements have common properties; so, such properties can be defined once in an *abstract* element, then they can be inherited by the elements needing them.

In order to explain such concepts, let us consider the case of the PFT formalism; Fig. 1 shows the elements of the PFT formalism, using a UML-like graphic language where each box indicates the name of an element, its properties and the results computable on it. In Fig. 1, the nodes of a PFT can be events or gates; for this reason, two abstract elements derive from the basic element *Node*. Such abstract elements are $EVENT$ and $GATE$; the first one collects the common properties of the event nodes: *Label* and *Description*. Four types of event node derive from $EVENT$: *BASIC_EVENT*, *INTERNAL_EVENT*, *TOP_EVENT*, *REPLICATOR_EVENT*; such elements are instantiable in a PFT model. A result named *Probability* is defined for TOP_EVENT. $BASIC\_EVENT$ has two additional properties (*Distribution_Type*, *Distribution_Parameter*) and one result (*Criticality*). $REPLICATOR\_EVENT$ has two

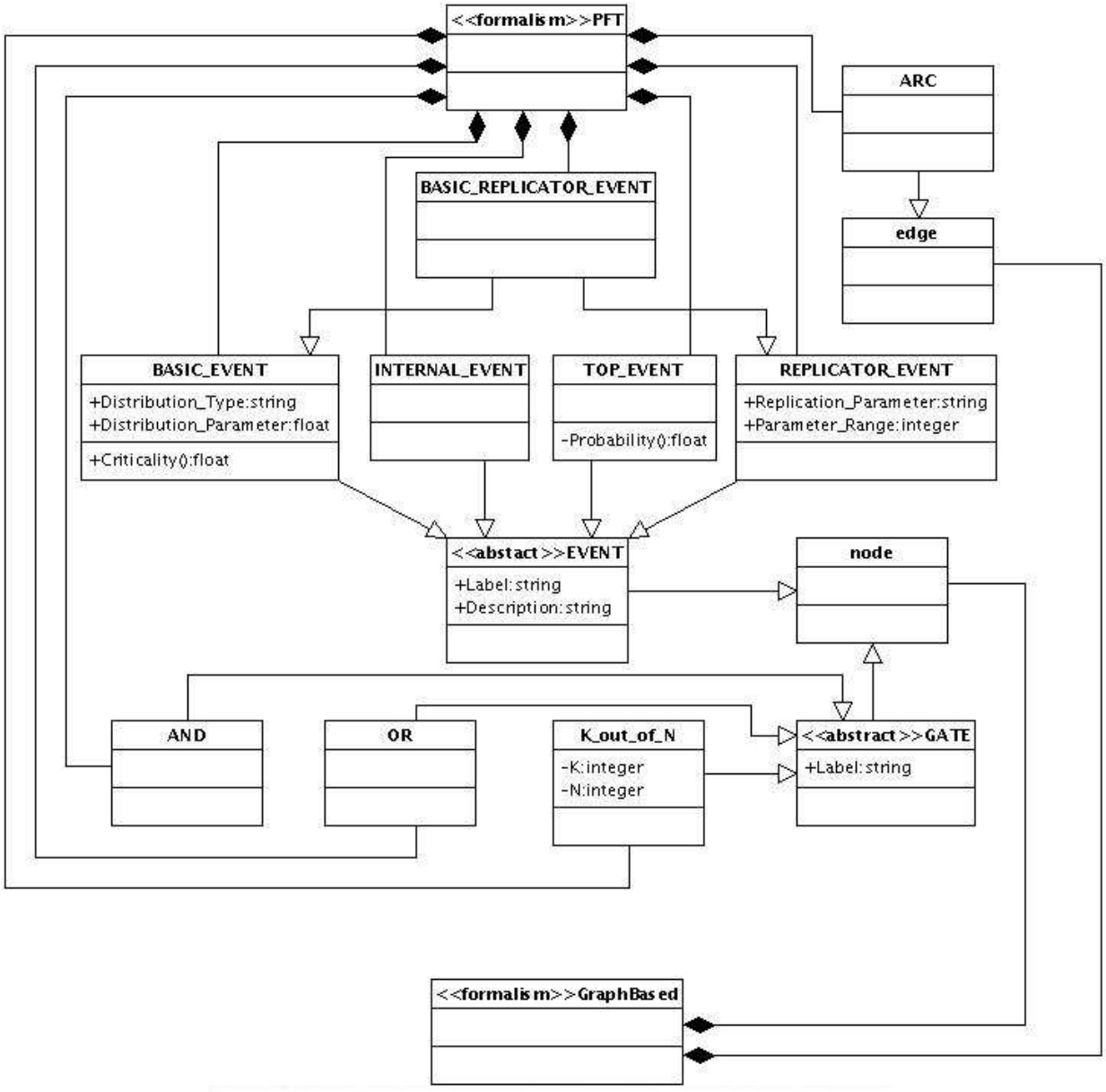

Figure 1: UML-like diagram of the simple formalism PFT.

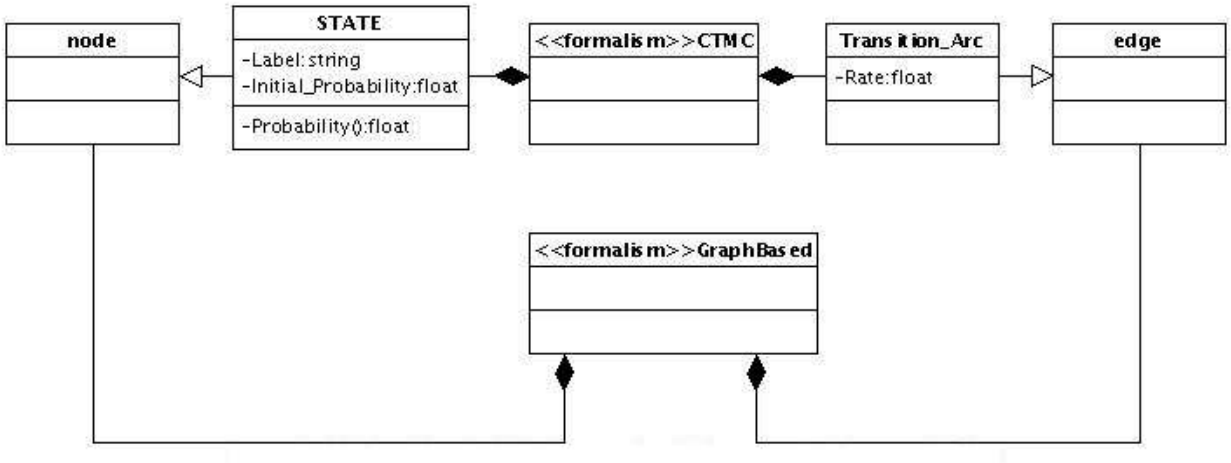

Figure 2: UML-like diagram of the simple formalism CTMC.

specific properties: *Replication_Parameter* and *Parameter_Range*. A further type of event node is present in the diagram in Fig. 1: *BASIC_REPLICATOR_EVENT*; such element inherits the properties and results of both *REPLICATOR_EVENT* and *BASIC_EVENT*.

The abstract element $GATE$ has one property: *Label*. Such property is inherited by all the types of gate deriving from $GATE$: AND, OR, $K\_out\_of\_N$. The last one has two specific properties: K and N. The type ARC concerns the connection arcs of the PFT; it derives from the basic element *Edge*.

The use of abstract elements may be useful also to set connection constraints among nodes. In the case of PFT, an arc can connect only an event to a gate or vice-versa; so, we can establish such constraint in the element ARC, by involving the abstract elements $EVENT$ and $GATE$; in this way, the constraint holds for any type of event or gate deriving from the abstract elements $EVENT$ and $GATE$.

CTMC is a simple formalism too: it is shown in Fig. 2. Its elements are $STATE$ and $TRANSITION\_ARC$; *Label* and $Initial\_Probability$ are the properties of $STATE$, while *Probability* is its result; *Rate* is the property of $TRANSITION\_ARC$.

4.2 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,

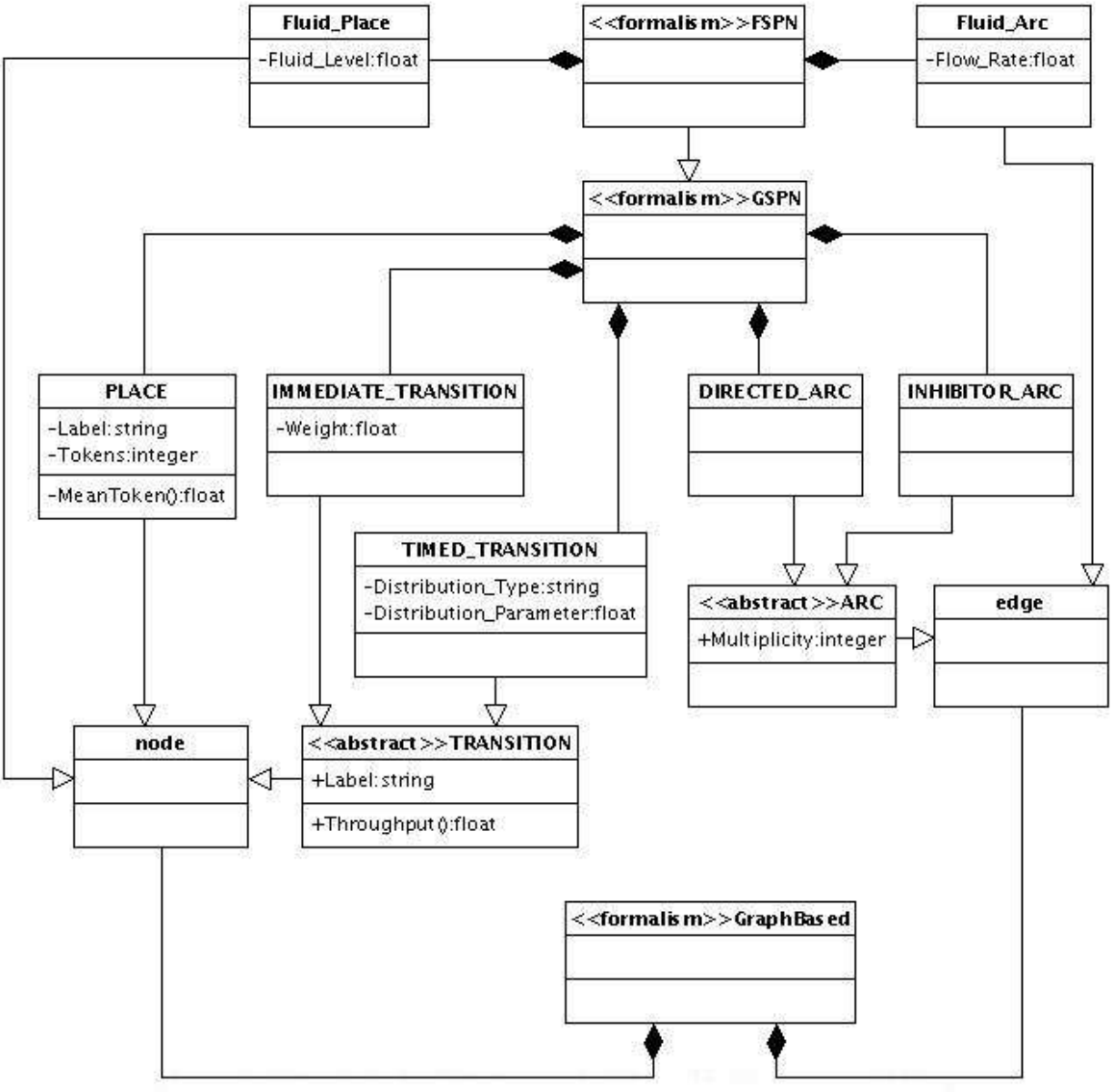

Figure 3: UML-like diagram of the derived formalism FSPN.

suppose that an element e ($e \neq e_0$) 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.

A case of derived formalism is FSPN derived from GSPN. Fig. 3 shows the elements of both the GSPN and FSPN formalisms using an UML-like graphic language. The GSPN elements $PLACE$ and $TRANSITION$ are nodes, since they derive from the basic element *Node*. $PLACE$ has two properties (*Label*, *Tokens*) and one result ($Mean\_Tokens$). $TRANSITION$ is an abstract element collecting the common property (*Label*) and the common result (*Throughput*) of the transitions. Two elements derive from $TRANSITION$: *IMMEDIATE_TRANSITION*, *TIMED_TRANSITION*; the first one has one property called $Weight$, while the second one has two specific properties: $Distribution\_Type$ and $Distribution\_Parameter$. In the GSPN formalism, two types of edge are defined: $DIRECTED\_ARC$ and $INHIBITOR\_ARC$; both of them derive from the abstract element ARC whose unique property is $Multiplicity$. ARC derives from the basic element *Edge*.

FSPN formalism is a derivation from the GSPN formalism: FSPN inherits the same elements of GSPN, with the addition of two new elements: $FLUID\_PLACE$ and $FLUID\_ARC$; both of them have one property: $Fluid\_Level$ and $Flow\_Rate$ respectively. $FLUID\_PLACE$ derives from *Node*, while $FLUID\_ARC$ derives from *Edge*.

4.3 Composed formalisms

In a multi-formalism model, we may 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. Moreover, a container model must indicate the way to solve each sub-model and to combine the results together providing the solution of the whole multi-formalism model.

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 *Container* and consisting of an element called $SOLVER$ which is a node, and two edges called $COMMUNICATION\_ARC$ and $SOLUTION\_ARC$. The property of the $SOLVER$ 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.

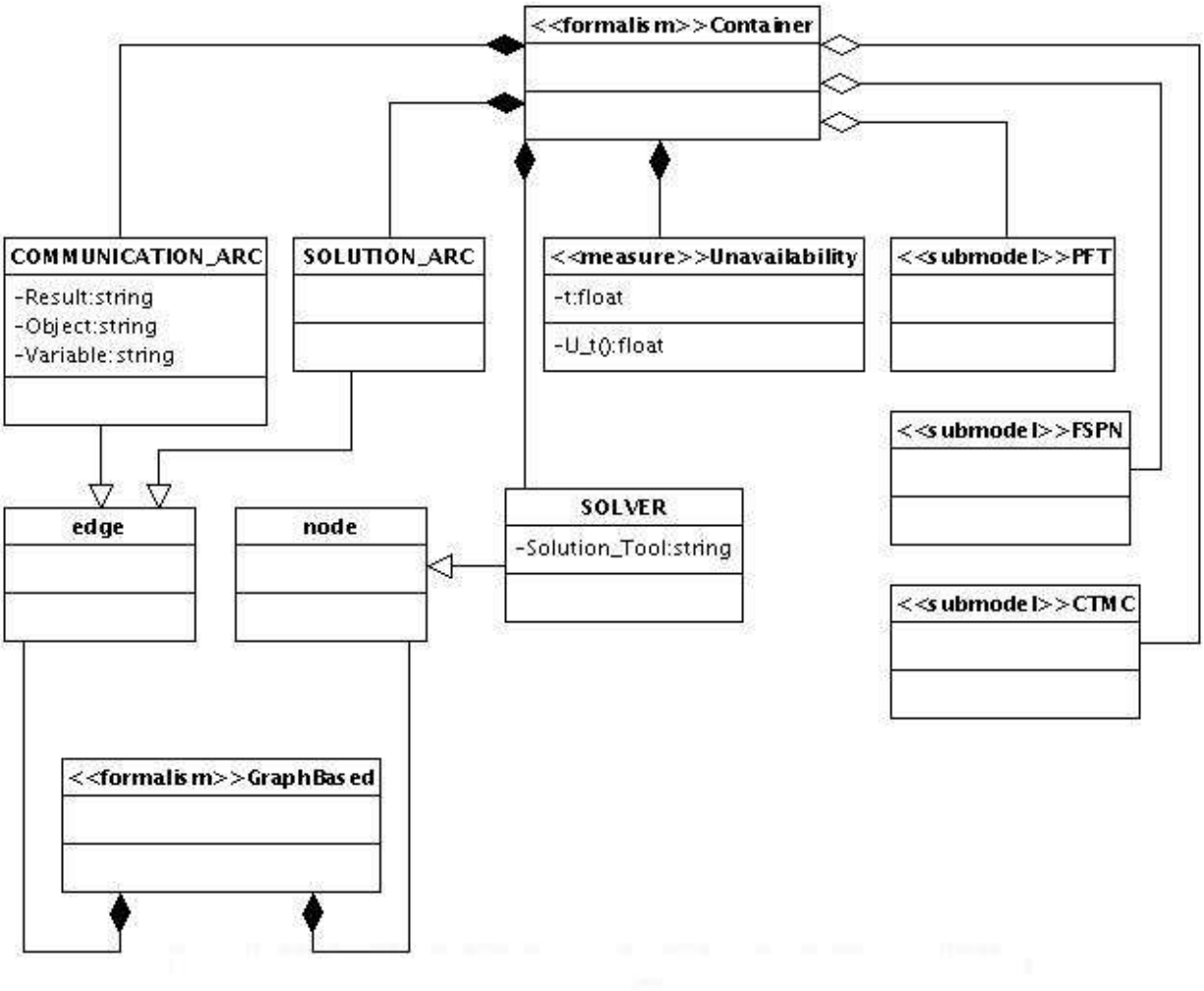

Figure 4: UML-like diagram of the composed formalism *Container*.

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 PFT, FSPN and CTMC.

However, the use of a container model with several sub-models, is not the only possible structure of a multi-formalism model; for this reason, a composed formalism may include other composed formalisms; in this way, we can have a hierarchy of formalisms organized in several levels.

4.4 *DNForGe*

The *Draw-Net tool* (model editor) 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 "by hand", 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* (*Draw-Net FOrmalism GEnerator* 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*.

Let us consider the composed formalism introduced in Sec. 4.3; the main window of *DNForGe* is named "*DNForGe: Composer*" (Fig. 5.a) and displays in a tree graphical structure the hierarchy of the involved formalisms; composed formalisms are indicated by a folder icon to express that they can include one or more formalisms. 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 this way, another window appears and is is named "*DNForGe: Formalism Elements*" (Fig. 5.b) and allows the user to add or remove elements (nodes, edges, measures) in a certain formalism. If the user selects an element of the formalism and decides to edit it, a further window appears and is called "*DNForGe: Element Editor*" (Fig. 5.c); here, the user can add, remove or edit the properties of the previously selected element.

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.

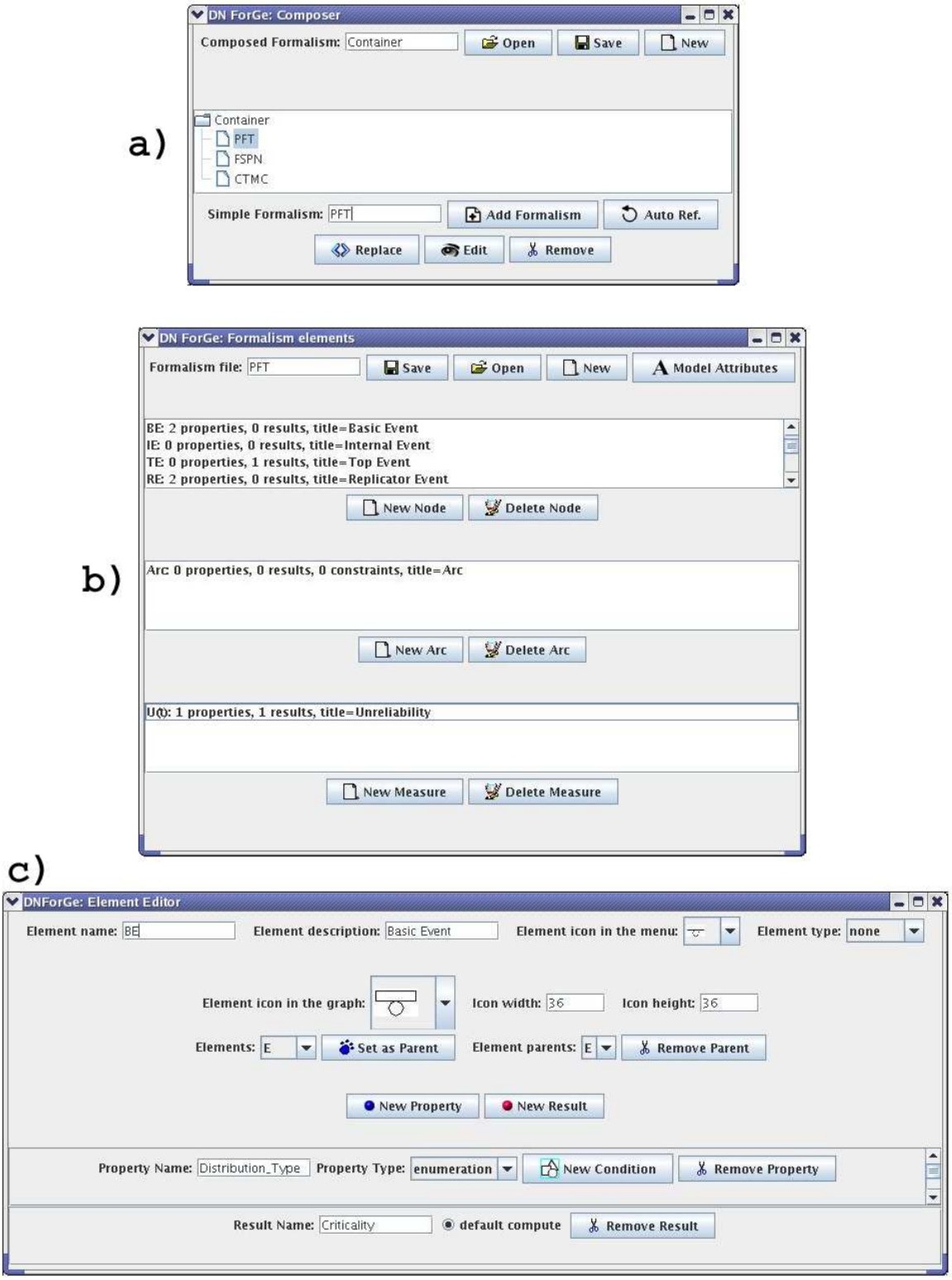

Figure 5: Screenshots of *DNForGe*: a) Composer; b) Formalism Elements; c) Element Editor.

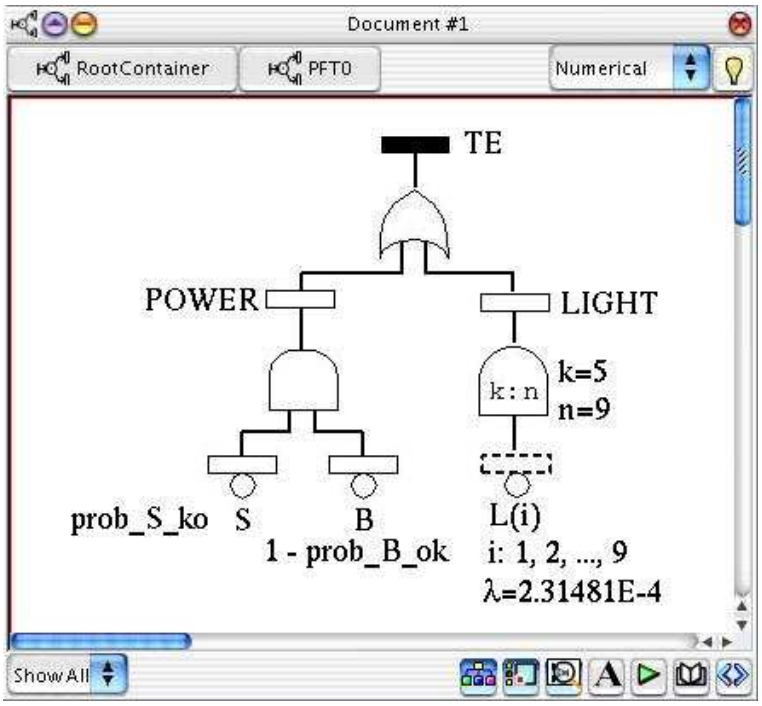

Figure 6: Submodel PFT_0.

5 Building models in the DMS

In this section the multi-formalism model representing the behaviour of the system proposed in Sec. 3, is described; the model is composed by a container model called *RootContainer*, and three sub-models: PFT_0, $FSPN_1$, $CTMC_2$. All of them are instances of the formalisms defined in Sec. 4. Each sub-model represent a specific features of the system behaviour, as mentioned at the end of Sec. 3. In this section, first, each sub-model is presented, then the container model is shown. Such multi-formalism model is drawn by means of the *Draw-Net tool*.

PFT model

PFT_0 is shown in Fig. 6 and respects the PFT formalism (Sec. 4.1); TE is an instance of TOP_EVENT and indicates the system failure. TE is the output of an instance of an OR gate whose input events are $POWER$ and $LIGHT$; so, TE happens if the event $POWER$ or the event $LIGHT$ occurs; both of them are instances of $INTERNAL\_EVENT$ in the PFT formalism; this means that they are the output of a gate.

$POWER$ models the absence of power supply and is the output of an instance of AND; the input of such gate are instances of $BASIC\_EVENT$ named S and B; they model the failure of the power supply

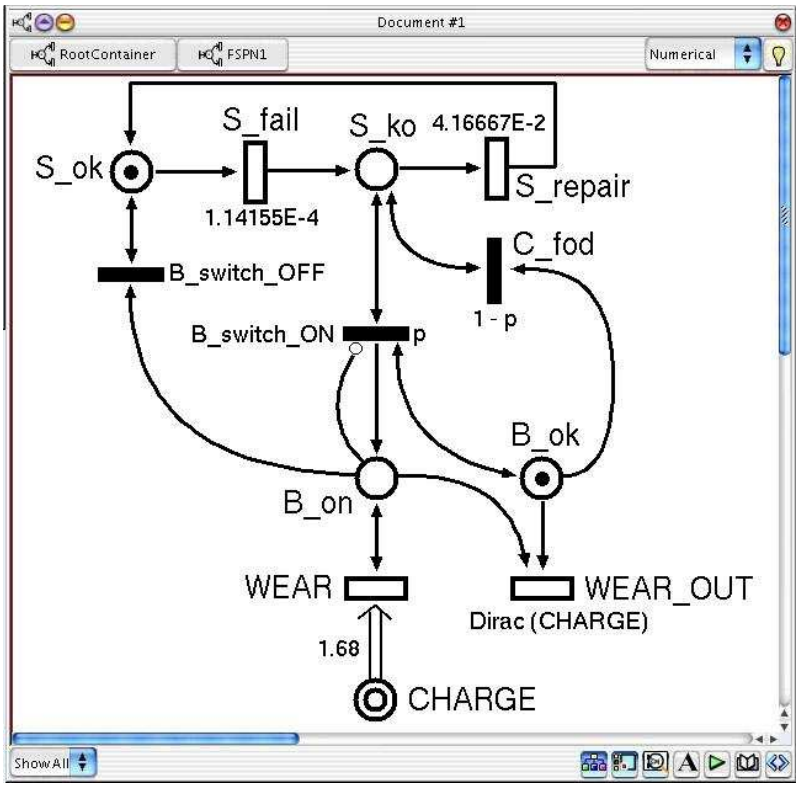

Figure 7: Submodel $FSPN_1$.

and of the battery respectively. They are stochastic events whose property *Distribution_Type* is set to "*Constant_Probability*" with the property *Distribution_Parameter* set to the expression *prob_S_ko* and $1 -$ *prob_B_ok* respectively. $POWER$ occurs if both S and B occur.

$LIGHT$ is the output of an instance of $K\_out\_of\_N$, with the properties K=5 and N=9; an instance of $BASIC\_REPLICATOR\_EVENT$ is the input of such gate; this instance is named $L(i)$ having i as *Replication_Parameter* and a *Parameter_Range* varying from 1 to 9. This means that $L(i)$ replaces 9 distinct instances of $BASIC\_EVENT$ with the same *Distribution_Type* ("*Negative_Exponential*") and with the same *Distribution_Parameter* (0.000231481, mentioned in Sec. 3). $L(i)$ models the failure of each of the 9 lamps. So, $LIGHT$ occurs when $K = 5$ of the $N = 9$ lamps fail.

All event and gates are connected together by means of instances of ARC respecting the constraint defined in the PFT formalism.

FSPN model

Fig. 7 shows the model $FSPN_1$ for the power supply, respecting the FSPN formalism (Sec. 4.2). *S_ok* and *S_ko* are instances of $PLACE$, so they can contain a discrete number of tokens. *S_ok* and *S_ko* model respectively the working and the failed condition of the power supply S; when one of these place is marked (contains one token), the relative condition is true, else it is false. Initially, *S_ok* contains one token ($Tokens = 1$), while *S_ko* is empty ($Tokens = 0$).

S_fail is an instance of $TIMED\_TRANSITION$; this transition models the failure of S, so it has to fire after a random period of time in order to move the token from the place *S_ok* to the place *S_ko*. For *S_fail*, *Distribution_Type="Negative_Exponential"* and *Distribution_Parameter* is equal to the failure rate of S, mentioned in Sec. 3. *S_repair* is another instance of $TIMED\_TRANSITION$ and models the repair of S by moving the token from the place *S_ko* back to *S_ok*; it has the same *Distribution_Type* of *S_fail* and its *Distribution_Parameter* is set to the repair rate of S.

All arcs in $FSPN_1$ are instances of $DIRECTED\_ARC$ and are used by the transitions to move tokens from a place to another. Another instance of $PLACE$ is *B_ok*; if such place is marked, the battery is not failed. Initially, for *B_ok*, $Tokens = 1$. The contemporary marking of *S_ko* and *B_ok*, enables the firing of two instances of $IMMEDIATE\_TRANSITION$: *B_switch_ON* and *C_fod*, modeling respectively the switch of the power supply from S to B, and the failure on demand of C. Such transitions are enabled at the same time; for this reason, the property $Weight$ establishes the probability to fire of each of them: for *B_switch_ON*, $Weight = p$; for *C_fod*, $Weight = 1 - p$.

If the transition *B_switch_ON* fires, *B_on* becomes marked; *B_on* is an instance of $PLACE$ and when it is marked, it models that the battery is supplying electric power. If instead, the transition *C_fod* fires, the token inside *B_ok* is removed; when *B_ok* is empty, the battery is failed.

Let us consider now $CHARGE$ as an instance of $FLUID\_PLACE$; this means that $CHARGE$ contains a continuous amount of fluid instead of a discrete number of tokens. Fluid places are useful to model the variation of continuous quantities, such as the charge level of the battery. For $CHARGE$, initially the property *Fluid_Level* is equal to 100, in order to model the complete charge of the battery. $WEAR$ is an instance of $TIMED\_TRANSITION$, enabled while the place *B_on* is marked, and connected to $CHARGE$ by means of an instance of $FLUID\_ARC$ having $Flow\_Rate = 1.68$; while $WEAR$ is enabled, some fluid is removed from $CHARGE$ with respect to the value of the property *Flow_Rate*. This models the gradual decrease of the charge of the battery while it supplies electric power. $WEAR\_OUT$ is an instance of $TIMED\_TRANSITION$; its properties are *Distribution_Type="Dirac"* and $Distribution\_Parameter = CHARGE$; this means that $WEAR\_OUT$ fires when the fluid level inside $CHARGE$ is equal to 0, in other words when the charge of the battery is null. The effect of the firing of this transition is the removal of the

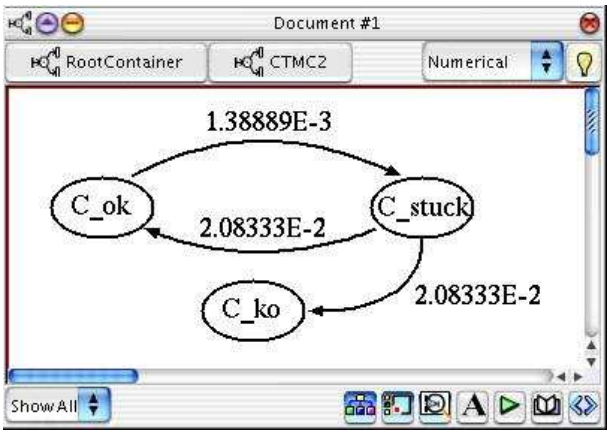

Figure 8: Submodel $CTMC_2$.

token from the place B_ok.

B_switch_OFF is another instance of $IMMEDIATE\_TRANSITION$; if B_on is marked and one token appears in S_ok due to the firing of S_repair, B_switch_OFF fires removing the token inside B_on; in this way, we model the switch of the power supply from B to S, when S is repaired.

CTMC model

$CTMC_2$ (Fig. 8) respects the CTMC formalism (Sec. 4.1) and shows the possible states of the controller C. C_ok is an instance of $STATE$ with the property $Initial\_Probability = 1$, so it is the initial state; from this state, the controller can turn to the instance of $STATE$ named C_stuck, by means of an instance of $TRANSITION\_ARC$ having $Rate = 0.00138889$; from C_stuck, C can turn back to C_ok by means of an instance of $TRANSITION\_ARC$ having $Rate = 0.0208333$, or to another instance of $STATE$ named C_ko by means of an instance of $TRANSITION\_ARC$ with the same previous value for the property $Rate$. C_ko is an absorbing state. C_stuck and C_ko have $Initial\_Probability = 0$. C_ok, C_stuck and C_ko represent respectively the working, stuck and failed condition of the controller C.

Container model

Fig. 9 shows the container model called $RootContainer$; it respects the $Container$ formalism (Sec. 4.3) and sets the interconnections among the sub-models PFT_0, $FSPN_1$ and $CTMC_2$, explained above. A measure named $Unavailability$ whose property is t (time), concerns the whole multi-formalism model and indicates the unavailability of the system at a given time t. Several instances of $COMMUNICATION\_ARC$ connecting sub-models, indicate the exchange of results among sub-models: in PFT_0 (Fig. 6), the property

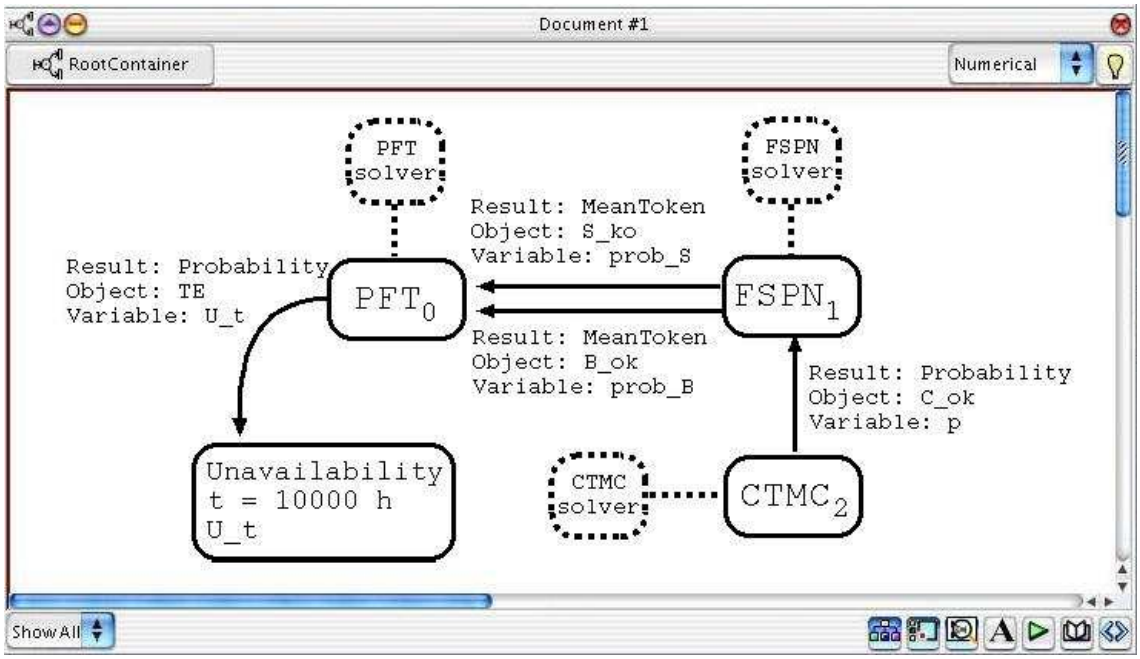

Figure 9: Container model.

Distribution_Parameter of the basic event S is set to the variable *prob_S_ko*, while *Distribution_Parameter* of the basic event B is set to $1 - prob\_B\_ok$; the properties of the arcs connecting $FSPN_1$ to PFT_0 indicate that *prob_S_ko* and *prob_B_ok* must be computed on $FSPN_1$ as the mean number of tokens present in the places called *S_ko* and *B_ok* 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.

In $FSPN_1$, the property *Weight* of the transition *B_switch_ON* is equal to the value of the variable p, while the same property of the transition *C_fod* is equal to $1 - p$. In *RootContainer*, the properties of the arc connecting $CTMC_2$ to $FSPN_1$ indicate that p must be computed on $CTMC_0$ as the probability of the state *C_ok*.

Finally, the measure *Unavailability* relative to the whole multi-formalism model, is connected to PFT_0 to indicate that it is equal to the probability of TE in PFT_0. Due to the connections among the sub-models in the container model (Fig. 9), the unavailability analysis of the system at time t, must follow these steps:

1. analyzing $CTMC_2$ returning the probability of being in state *C_ok* at time t;

2. setting p to such probability in $FSPN_1$;

3. analyzing $FSPN_1$ returning the mean number of tokens inside the places S_ko and B_ok;

4. setting $prob\_S\_ko$ and $prob\_B\_ok$ to such values, in PFT_0;

5. analyzing PFT_0 returning the probability of TE at time t;

6. setting U_t to such value, in the container model; U_t is the system *Unavailability* at time t.

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

5.1 *Draw-Net tool*

One of the most important components of the DMS framework is the graphical user interface (simply referred as the *Draw-Net Tool*). It is a standard graphical user interface written in Java. The *Draw-Net tool* can read formalisms written using the DDL and saved using the XML-interchange formats described in Sec. 2.4. The choice of the Java language allows the interface to be easily ported into a wide variety of operating systems and architectures. The GUI exploits the *Java reflection API* to dynamically load its components. All the classes used to implement the user-interaction features are listed in an XML file and dynamically loaded at run-time. This makes also the GUI easily customizable and extensible.

The standard features of the GUI are its multi-document interface, layers support, the possibility to add many kind of graphical annotations. Arcs are drawn using Bezier curves, and can be broken if needed. It can include images, and it can export SVG files to produce high quality graphical representations of the model.

A structure panel allows the possibility to nest sub-models written in other formalisms. Arcs can then connect primitives from the enclosing model to its sub-model.

A screenshot of the *Draw-Net tool* can be seen in Fig. 10; other screenshots can be found on the *Draw-Net* web page [20].

5.1.1 Software architecture of the *Draw-Net tool*

The *Draw-Net tool* is designed in to allow customization and extensibility without need to recompile the source code of the tool. This is very important for a tool that wants to support a wide variety of formalisms. Many formalisms in fact require different graphical support. For example Bayesian Networks may require the user to draw tables and to use them to assign probability to the possible output, conditioned to the various combinations of inputs. The extensibility is achieved using three concepts: *panels*, *states* and *commands*.

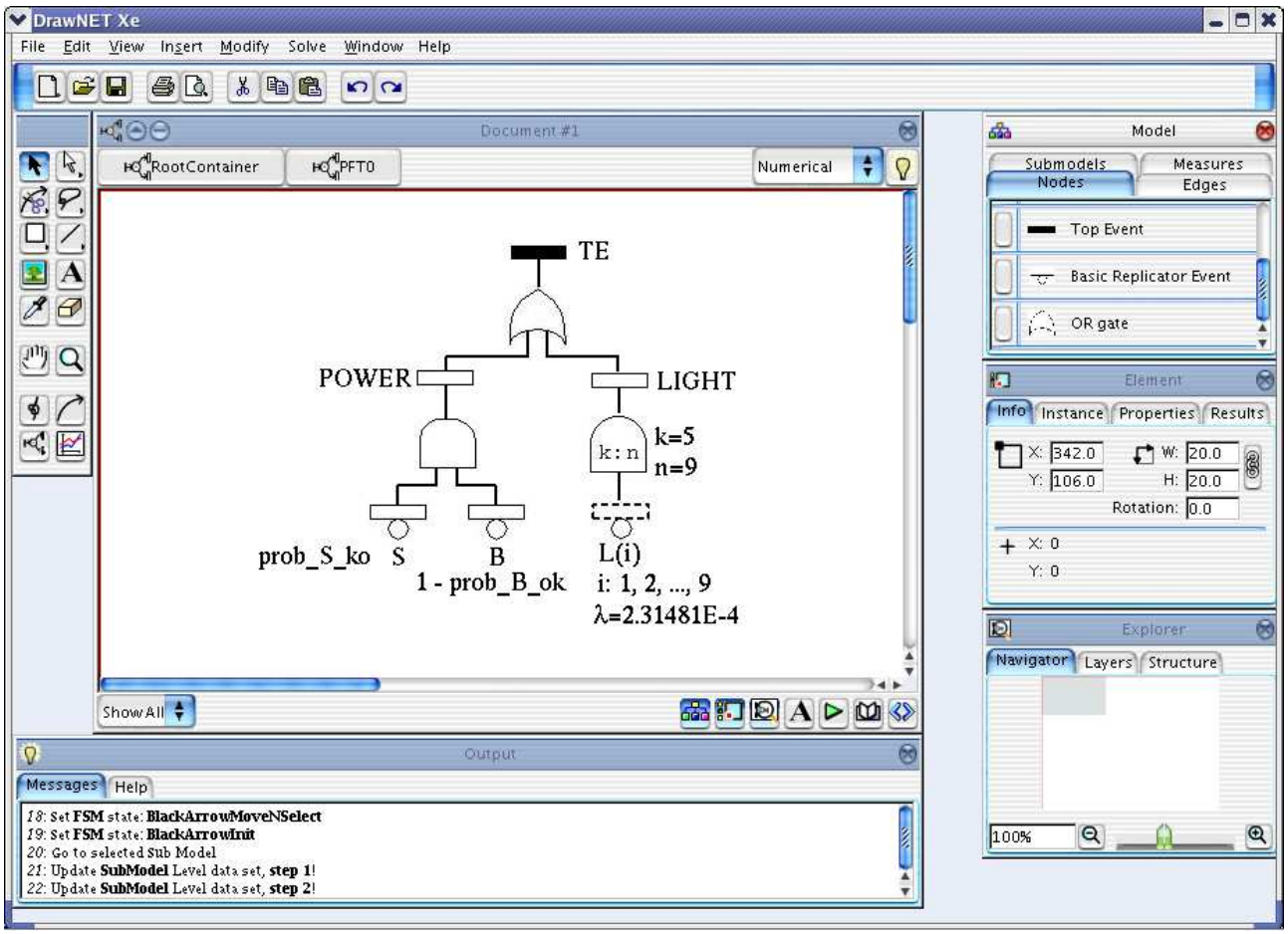

Figure 10: Screenshot of the *Draw-Net tool*.

Panels represent internal windows of the GUI. They can be used to provide functionalities, like new drawing tools, drawing aid or to present customized input panels, specific for elements of a particular formalism.

States define the way in which user interface works. The drawing in the main window is regulated by a Finite State Machine (FSM). Each drawing tool selected by the user puts the FSM in a different state. The actions done by the mouse over the main window causes *FSM events*. Each state defines how to react to these events, and possibly a new state to jump to. This allows complex interactions like the one required to draw arcs using Bezier curves.

Commands produce changes to the model or to the tool. Every action done by the user is implemented by executing a particular command. The *Draw-Net tool* provides a standard way to record changes made by a command. This allows the implementation of a undo / redo and history feature independent from the nature of a particular command. Commands are divided into four levels, depending on the type of action they perform: *Tool*, *Frame*, *SubModel* and *Element*. Tool commands perform actions relative to the entire GUI and not related to any model, like opening a new window or showing a new panel. Frame commands perform actions related to a particular model, like saving it to disk or passing it to a solver. SubModel commands perform actions related to a specific sub-model, like adding or renaming an element. Element commands operate on the single elements: for example, they assign properties or show results.

The key feature of the extensibility of the GUI is that panels, states and commands can be added without needing to recompile the GUI. They are all created by extending the corresponding java base classes, and are dynamically loaded using Java introspection API. In particular the name of the classes that implement the required panels, states and commands are stored in a configuration file written in XML. The *Draw-Net tool* dynamically creates instances of these classes during its startup. An example of this configuration file is presented in the Appendix.

6 Native solvers

The DMS framework provides a standard way to define single or multi-formalism models in a user-friendly way. However, model definition is usually only meaningful if solver applications that can compute results on these models, exist. In this section, we will briefly present some of the solution components that have already been included in the DMS framework.

FSPNedit [21] is a tool based on the DMS and allows the solution of FSPNs. It can provide model analysis using both numerical techniques and discrete event simulation.

Another tool [22] based on the DMS, concerns the analysis of an extension of PFTs called *Dynamic Repairable Parametric Fault Trees* (DRPFT) [23, 24, 25]; such formalisms includes *dynamic gates* to model dependencies among failure events, and *repair boxes* to model repair processes. The analysis of DRPFTs

involves two methods: the combinatorial solution and the state space solution.

There is another tool for the analysis of Fault Tree extensions using the *Draw-Net tool* as graphical interface; such tool implements the conversion of simpler *Dynamic Fault Trees* (DFT) [26, 27] in Dynamic Bayesian Networks (DBN) [28, 29, 30]. The use of BNs allows to compute new measures on DFTs. Also a DBN solver has been based on the DMS.

Such solvers concern single models; future developments will regard the study of generalized solution methods for multi-formalism models.

6.1 Analysis of the running example

In this section, we perform the transient analysis of the multi-formalism model described in Sec. 5 and representing the behaviour and the failure mode of the system described in Sec. 3. We are interested in computing the *Unavailability* of the system after $1000h$.

We use the *Sharpe* tool to solve $CTMC_2$ and PFT_0, while we use *FSPNEdit* to solve $FSPN_1$. According to the analysis steps given by the container model and indicated at the end of Sec. 5, these are the intermediate results obtained on the sub-models for a mission time equal to $1000h$:

1. the analysis of $CTMC_2$ returns that the probability of the state C_ok is equal to 0.496608.
2. In $FSPN_1$, p is set to 0.496608.
3. The analysis of $FSPN_1$ returns that
   - the mean number of tokens inside the place S_ko is equal to 0.002732;
   - the mean number of tokens inside the place B_ok is equal to 0.892103.
4. In PFT_0,
   - *prob_S_ko* is set to 0.002732;
   - *prob_B_ko* is set to 0.892103.
5. The analysis of PFT_0 returns that the probability of TE is equal to 0.022751.
6. In the container model, U_t is set to 0.022751.

Thus, from the analysis of the multi-formalism model for a mission time of $1000h$, we obtain that the *Unavailability* of the system at that time (U_t), is equal to 0.022751.

7 Conclusions and future work

The DMS is an open and extensible framework supporting the multi-formalism multi-solution approach to the modeling of systems: it allows to model different aspects of a system with the most appropriate formalism and analyze it through a solution process based on one or more solvers.

The basic ideas behind the DMS were already introduced in its first implementations, but they have evolved significantly: with respect to the previous version [9], the DMS includes now *DNlib*, *DDL*, *DNForGe*; moreover, the *Draw-Net tool* has been completely redesigned, with the addition of new facilities increasing its extensibility.

Future work will concern the implementation of multi-solution in the DMS; in particular, future work will regard the integration of the available native solvers together with the construction of filters to get interfaced with new solvers.

References

[1] 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.

[2] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Möbius Modeling Tool. In *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001.

[3] T. Courtney, D. Daly, S. Derisavi, V. Lam, and W. H. Sanders. The Möbius Modeling Environment. In *Tools of the 2003 Illinois International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communication Systems*, volume research report no. 781/2003, pages 34–37, Universitat Dortmund Fachbereich Informatik, 2003.

[4] T. Courtney, D. Daly, S. Derisavi, S. Gaonkar, M. Griffith, V. Lam, and W. H. Sanders. The Möbius Modeling Environment: Recent Developments. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST)*, Twente, The Netherlands, September 2004.

[5] G. Ciardo and A. S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *Proc. of the 7th Int. Workshop on Petri Nets and Performance Models*, pages 41–43, Saint Malo, France, June 1997.

[6] R. Sahner and K. S. Trivedi. Reliability modeling using SHARPE. *IEEE Transactions on Reliability*, R-36:186–193, 1987.

[7] 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.

[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] V. Vittorini, G. Franceschinis, M. Gribaudo, M. Iacono, and N. Mazzocca. DrawNet++: Model objects to support performance analysis and simulation of complex systems. In *Proceedings 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS 2002)*, pages 233–238, London, 2002. Springer Verlag - LNCS, Vol 2324.

[10] 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.

[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] A. Bobbio, G. Franceschinis, R. Gaeta, and G. Portinale. Parametric fault tree for the dependability analysis of redundant systems and its high-level Petri net semantics. *IEEE Transactions on Software Engineering*, 29(3):270–287, March 2003.

[13] G. Franceschinis, R. Gaeta, and G. Portinale. Dependability Assessment of an Industrial Programmable Logic Controller via Parametric Fault-Tree and High Level Petri Net. In *Proc. 9th Int. Workshop on Petri Nets and Performance Models*, pages 29–38, Aachen, Germany, Sept. 2001.

[14] A. Bobbio, D. Codetta-Raiteri, Massimiliano De Pierro, and G. Franceschinis. Efficient Analysis Algorithms for Parametric Fault Trees. In *Proceedings of the Workshop on Techniques, Methodologies and Tools for Performance Evaluation of Complex Systems*, pages 91–105, Turin, Italy, September 2005.

[15] M. Gribaudo, M. Sereno, and A. Bobbio. Fluid Stochastic Petri Nets: An Extended Formalism to Include non-Markovian Models. In *8-th International Conference on Petri Nets and Performance Models - PNPM99*, pages 71–82. IEEE Computer Society, 1999.

[16] M. Gribaudo, A. Bobbio, and M. Sereno. Modeling physical quantities in industrial systems using Fluid Stochastic Petri Nets. In *Proceedings 5-th International Workshop on Performability Modeling of Computer and Communication Systems*, pages 81–85, 2001.

[17] M. Gribaudo, M. Sereno, A. Horvath, and A. Bobbio. Fluid Stochastic Petri Nets augmented with flush-out arcs: Modelling and analysis. *Discrete Event Dynamic Systems*, 11(1/2):97–117, 2001.

[18] K. Trivedi. *Probability & Statistics with Reliability, Queueing & Computer Science applications*. Wiley, II Edition, 2001.

[19] M. Ajmone-Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. J. Wiley and Sons, 1995.

[20] http://www.draw-net.com. Draw-Net web page.

[21] 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.

[22] D. Codetta-Raiteri. Development of a Dynamic Fault Tree solver based on Colored Petri Nets and graphically interfaced with DrawNET. Technical Report TR-INF-2003-10-06-UNIPMN, Dipartimento di Informatica, Università del Piemonte Orientale, October 2003.

[23] A. Bobbio and D. Codetta-Raiteri. Parametric Fault-trees with dynamic gates and repair boxes. In *Proc. Reliability and Maintainability Symposium*, pages 459–465, Los Angeles, CA USA, January 2004.

[24] D. Codetta-Raiteri. Parametric Dynamic Fault Tree and its Solution through Modularization. In *Supplemental Volume of the International Conference on Dependable Systems and Networks*, pages 157–159, Florence, Italy, June 2004.

[25] A. Bobbio, D. Codetta-Raiteri, M. De Pierro, and G. Franceschinis. System Level Dependability Analysis. In M. Violante M. Sonza Reorda, Z. Peng, editor, *System-level Test and Validation of Hardware/Software Systems*, volume 17 of *Springer Series in Advanced Microelectronics*. Springer, 2005.

[26] J. B. Dugan, S. J. Bavuso, and M. A. Boyd. Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems. *IEEE Transactions on Reliability*, 41:363–377, 1992.

[27] R. Manian, D. W. Coppit, K. J. Sullivan, and J. B. Dugan. Bridging the Gap Between Systems and Dynamic Fault Tree Models. In *Proceedings Annual Reliability and Maintainability Symposium*, pages 105–111, 1999.

[28] A. Bobbio, L. Portinale, M. Minichino, and E. Ciancamerla. Improving the Analysis of Dependable Systems by Mapping Fault Trees into Bayesian Networks. *Reliability Engineering and System Safety*, 71:249–260, 2001.

[29] S. Montani, L. Portinale, and A. Bobbio. Dynamic Bayesian Networks for Modeling Advanced Fault Tree Features in Dependability Analysis. In *Advances in Safety and Reliability (ESREL 2005)*, volume 2, pages 1415–1422. Balkema, 2005.

[30] S. Montani, L. Portinale, A. Bobbio, M. Varesio, and D. Codetta-Raiteri. DBNet, a tool to convert Dynamic Fault Trees to Dynamic Bayesian Networks. Technical Report TR-INF-2005-08-02-UNIPMN, Dipartimento di Informatica, Università del Piemonte Orientale, August 2005.

Appendix: XML based interchange format

The content of this appendix supports the explanation of the concepts reported in this paper. In this appendix, we present the FDL definition files generated by *DNForGe* for the CTMC formalism and for the Container formalism, the MDL file generated by the *Draw-Net tool*, and an excerpt of the *Draw-Net tool* configuration file that dynamically loads panels, states and commands of the GUI.

FDL for the CTMC formalism

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE fdl SYSTEM '../../dtd/fdl.dtd'>
<fdl main="CTMC">
  <include src="base/GraphBased.fdl" />
  <include src="base/Instantiable.fdl" />
  <elementType name="CTMC" >
    <parent ref="GraphBased" />
    <parent ref="Instantiable" />
    <elementType name="STATE" >
      <parent ref="Node" />
      <propertyType name="Label" type="string" default="?" />
      <propertyType name="Initial_Probability" type="float" default="0.1" />
    </elementType>
    <elementType name="Transition_Arc" >
      <parent ref="Edge" />
      <propertyType name="Rate" type="float" default="0.1" />
    </elementType>
  </elementType>
</fdl>
```

FDL for the Container formalism

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE fdl SYSTEM '../../dtd/fdl.dtd'>
<fdl main="Container">
  <include src="base/GraphBased.fdl" />
  <include src="base/Instantiable.fdl" />
  <include src="PetriNets/FSPN/FSPN.fdl" />
  <include src="FT/PFT/PFT.fdl" />
  <include src="MC/CTMC/CTMC.fdl" />

    <elementType name="Container" >
      <parent ref="GraphBased" />
```

```
      <parent ref="Instantiable" />
        <elementType name="SOLVER" >
          <parent ref="Node" />
            <propertyType name="Solution_Tool" type="string" default="?" />
        </elementType>
        <elementType name="COMMUNICATION_ARC" >
          <parent ref="Edge" />
          <propertyType name="Result" type="string" default="?" />
          <propertyType name="Object" type="string" default="?" />
          <propertyType name="Variable" type="string" default="?" />
        </elementType>
        <elementType name="SOLUTION_ARC" >
          <parent ref="Edge" />
        </elementType>

        <elementTypeRef ref="PFT" />
        <elementTypeRef ref="FSPN" />
        <elementTypeRef ref="CTMC" />
    </elementType>
</fdl>
```

MDL for the model

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<mdl fdl="Multiformalism/Container.fdl" main="RootContainer">
  <Container id="PowerPlant">
    <FSPN id='FSPN1' visibility='false' Title=''>
      <PLACE id='S_ok' visibility='false' Tokens='1' Label='S_working'/>
      <PLACE id='S_ko' visibility='false' Tokens='0' Label='S_failed'/>
      ....
    </FSPN>
    <CTMC id='CTMC2' visibility='false' Title=''>
      <STATE id='Cok' visibility='false' Label='C_working' Initial_Probability='1'/>
      <Transition_Arc id='Arc0' visibility='false' from='Cok' to='Cstuck' Rate='0.00138889'/>
       ....
    </CTMC>
    <PFT id='PFT0' visibility='false' Title=''>
      <TOP_EVENT id='TE' visibility='false' Label='System'/>
        <OR id='OR' visibility='false' Label='or1'/>
        <INTERNAL_EVENT id='POWER' visibility='false' Label='Power Supply'/>
        ....
    </PFT>
    <SOLVER id="FSPNsolver" Solution_Tool="FSPNedit"/>
```

```
    <SOLUTION_ARC id="Arc0" from="FSPN1" to="FSPNsolver"/>
         ....
  </Container>
</mdl>
```

The *Draw-Net tool* configuration file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE fdl SYSTEM "gdf.dtd">
<gdf>
<!-- States of the FSM -->
<FSMstate class="FSMStates.FSM_BlackArrowInit"/>
<FSMstate class="FSMStates.FSM_BlackArrowMoveNSelect"/>
<FSMstate class="FSMStates.FSM_BlackArrowMove"/>
....

<!-- Commands of the GUI -->
<command class="Commands.Tool.CMD_New"/>
<command class="Commands.Frame.CMD_Save"/>
<command class="Commands.SubModel.CMD_ZoomFact">
     <parameter type="integer" value="25"/>
</command>
....

<!-- Panels -->
<panelWindow name="Functions" caption="Functions"
icon="icons/Function.gif"
x="916" y="180" w="216" h="180" visible="false">
    <panelElement name="Transform" caption="Transform"
     class="Panels.FunctionPanel.DrawNETTransformPanel"/>
    <panelElement name="Align" caption="Align"
     class="Panels.FunctionPanel.DrawNETAlignPanel"/>
...
</panelWindow>

</gdf>
```