DrawNET++: model objects to support performance analysis and simulation of systems

G. Franceschinis[1], M. Gribaudo[2], M. Iacono[3], N. Mazzocca[3], and V. Vittorini[4]

[1] Di.S.T.A. ,Univ. del Piemonte Orientale, AL, Italy. giuliana@mfn.unipmn.it
[2] Dip. di Informatica, Univ. di Torino, Italy. marcog@di.unito.it
[3] Dip. di Ingegneria dell'Informazione, Seconda Univ. di Napoli, Italy
{mauro.iacono,nicola.mazzocca}@unina2.it
[4] D.I.S., Univ. degli Studi di Napoli Federico II, Italy. vittorin@unina.it

Abstract. This paper describes DrawNET++, a prototype version of a model design framework based on the definition of model objects. DrawNET++ provides a graphical front-end to existing performance tools and a practical mean to study compositionality issues in multi-formalism environments. The object oriented features of DrawNET++ provide a flexible architecture for structuring complex models.

Keywords: Model Objects, Model composition, Performance Analysis Framework.

1 Introduction

The complexity of nowadays computer and communication systems calls for suitable software tools to support design decisions. As pointed out in [8] there is a need for *open* frameworks and software environments able to embed and integrate different modeling formalisms and solution/simulation techniques.

The philosophy behind the tool we are proposing in this paper goes in the direction of providing a framework where different formalisms can be easily defined, extended, and composed, and models descriptions can be exported in XML form. The type of formalisms supported by the tool are those for which it is natural to define a representation in terms of a graph. The elements of the graph can be (sub)models. The tool supports compositional approaches to model construction, facilitating model structuring and (sub)model reuse in a style inspired by the Object Oriented (OO) paradigm.

Since new formalisms can be invented and easily integrated in the tool without any programming effort, domain specific formalisms can be defined: in this case the nodes may represent domain specific submodels expressed in some underlying formalism by an expert model designer, and presented to the final user as black boxes with proper interface and connectors.

The framework can exploit existing or newly created analysis/simulation engines by providing the model description as well as the description of performance indices to be computed in XML. Filters (e.g. XSL styles) need to be developed to

transform the XML model and performance indices representation in a format suitable for the specific analysis/simulation engine. Following this approach we believe that different solvers might be plugged-in as they become available at the (hopefully reasonable) price of writing proper filters.

This approach differs substantially from that advocated in the Möbius project [8] where new formalisms, composition operators and solvers are actually implemented within a unique comprehensive tool. In Möbius all formalisms are required to be compatible with a predefined general framework [5].

The experiments done so far with DrawNET++ have allowed us to experiment with a Fault Tree formalism (Parametric Fault Trees), to study compositionality of Stochastic Well Formed Net (SWN) models through the *algebra* tool [2] included in *GreatSPN* [3], and to mix the two things.

In Sec. 2 the features of DrawNET++ are presented. In Sec. 3 we discuss a possible approach for connecting DrawNET++ with existing model analyzers.

2 DrawNET++ features

Defining formalisms and models. *DrawNET* is a tool that can be used to rapidly develop user interfaces for performance evaluation engines: it provides a GUI to any graph-based formalism defined by the user such as queueing networks, Petri Nets, Fault trees, Bayesian networks and many more.

A formalism defines the kinds of nodes and edges models may include. Nodes, edges, and formalisms themselves are all *elements*, each with an associated name and set of *properties*. A property is a pair $(name, value)$ used to specify a characteristic of a particular element. Since also formalisms are elements, they can have some properties used to define attributes of the whole model. Edges can connect elements, and have an associated set of *constraints* that tell which kind of elements may be connected by that edge, and how many edges of a given type can be connected to a given element. An edge can connect not only two basic nodes, but also other edges, or (sub)models.

Formalisms and models are described in DrawNET by means of two different *data definition languages*[7]. Formalisms are specified by using a custom XML dialect called **FDL**. This dialect specifies which types of nodes and edges a formalism may have. Properties may be included into the tags that specify nodes and edges. Models are specified by using another custom XML dialect called **GDL**: models are collections of nodes, edges and sub-models satisfying a given FDL specification.

All elements of a model have a *visibility* attribute, which allows to hide internal details of sub-models. Edges can connect any element of the model, and any *visible* element of its sub-models.

Sub-models specified by means of distinct formalisms can coexist to build a more complex model. The only restriction is imposed on connections: visible elements of a sub-model (or the sub-model itself) may be connected to elements of the container model, only if suitable edges exist.

Specifying model objects. The DrawNET++ tool can be used as a support to an OO approach to performance model construction, as explained hereafter. Some extensions that have been implemented to this purpose are described at the end of this section.

Three levels can be defined in designing compositional models:
• **model class**: can be defined as a formalism, expressed though an FDL specification (e.g. BlackBox.xml, Machine.xml and FaultyMachine.xml in Fig.1);
• **model object**: it belongs to a given model class, is defined by the designer creating a model in DrawNET++, and is exported by the tool as an XML document following the GDL specification (e.g. the XML description of FaultyMachine indicated as a Template Model Object in Fig.1);
• **instance object**: it is an instance of a model object and it is expressed by means of the resulting XML specification exported by DrawNET. An instance object is created by the end user when building a complex system model (e.g. Robot1 or Cell1 in Fig.1). Several instances of a given model object can be included as *submodels* within a more complex model at a higher level.

A model object represents an abstraction of a system component: it comprises an **interface** and an **inner set of data**. The interface includes the relevant characteristics for the interaction of the modeled component with other components, that is the information that needs to be exchanged between the model and the environment in which it is integrated (*ExpInterface*,*ImpInterface*) or the parameters that need to be specified to complete the component description when it has some parametric part (*InstInterface*). It also includes the information about the interface points visible to the external world and usable for actually composing a model instance with other model instances (*InputInterface*, *OutputInterface*). The visibility property of both the *InputInterface* and *OutputInterface* elements should be set to true, so that they may be used to compose the model instance with other elements. This however does not imply that the properties of input/output interfaces will also be visible, only the interface elements will be visible so that they can be connected with external world elements through appropriate edges.

The inner set of data (the *Internal* node) consists of the internal details of the component behavior: it might include a graph directly drawn with DrawNET, or it may be a pointer to a model defined within another tool (e.g., a Stochastic Petri Net model defined in GreatSPN): in this paper we shall consider only the second option. In any case the inner set of data will not be visible when the model is included as a submodel in a more complex structure.

Up to now we have seen how model classes can be defined, and how model objects can be defined and instantiated: these features are present also in other existing tools (for example Tangram-II[4]). Our proposal goes one step further and implements some additional OO features described hereafter.
Inheritance of model specifications: inheritance between model classes can be defined at the formalism level since DrawNET++ allows to define model class hierarchies. A new FDL specification may derive from an existing FDL (e.g. see BlackBox.xml, Machine.xml and FaultyMachine.xml in Fig.1). In this context,

inheritance is a form of model reusability, in which new model classes are created from existing ones by absorbing their elements (nodes and/or edges) and overriding some of them or extending their properties.
Templates: Once a model object is defined, the property values of its nodes become hidden, however it is possible to define *template model objects* by declaring some properties as *parametric* so that they can be modified after instantiation (e.g. parameter *Fault Rate* in the FaultyMachine template model object).
Weak Aggregation: An object may be *aggregated* to other objects by means of proper composition operators. By now we have just explored a simple synchronization operator that is implemented as an edge that connects an *InputInterface* node with an *OutputInterface* node and vice-versa. More complex operators might be needed, that connect more than two elements, requiring the definition of "connection nodes" (for example to compose several output interfaces to several input interfaces, through a Cartesian product operator). The complete model of a system is obtained by instantiating and aggregating model objects as shown in Fig.1 for a simple FMS model.
Strong Aggregation: new model objects may be created that embed other model objects. When a new model is created by composing submodels, it is possible to exploit an information hiding feature of DrawNET++ to obtain strong aggregation. For example, in Fig.1 a model object *Cell* is created by aggregating a machine and a buffer, and hiding the *OutputInterface* of the machine and the *InputInterface* of the buffer so that they are no longer accessible when a *Cell* instance is included in a higher level model.

3 Performance analysis of DrawNET++ models

The work presented in this paper stemmed from the need of defining a framework for the compositional construction of SWN models using a library of (domain specific) models and an intuitive composition method based on black boxes and connectors [6, 1]. Although the framework can be used in a more general setting, for space reasons this section will be focused on this specific use of DrawNET++, and in particular on how it can be interfaced with GreatSPN.

The steps that allow to obtain performance results out of a DrawNET++ model similar to that shown in Fig. 1 are: (1) generate the (sub)model description files accepted by GreatSPN, (2) invoke the GreatSPN composition tool *algebra*, (3) invoke GreatSPN analysis/simulation facilities, (4) retrieve the performance results from the files generated by GreatSPN and save them in a form that can be read and displayed by DrawNET++.
Generation of SWN models for GreatSPN: In the model class descriptions of Fig 1 two properties allow to declare the formalism and associated analysis tool for the model objects in that class: in our case they are set to SWN and GreatSPN respectively. In this case a GreatSPN description of each *basic* model object behavior already exists: the part that must be implemented in a post-processor integrated into DrawNET++ is (1) the automatic generation of the connection structure, in the form of a GreatSPN labeled SWN model, (2) the

instantiation of the subnets represented by the DrawNET++ submodels, (3) the automatic generation of a script for calling *algebra* on the proper submodels, finally getting the complete composed model, ready for being solved/simulated. **Retrieving results:** the results that can be obtained by analysis and/or simulation are dependent on the specific tool used. A simple language as been defined to implement the communication of results , called Result Description Language (RDL). Based on the RDL, an XML Result Data Specification (RDS) defines the performance results by pairs (name, value). The RDS is used to back-propagate the information from a simulation/solution engine to the GUI: at the end of a simulation/solution process the post-processor retrieves the results and makes them available to DrawNET++ by filling a proper RDS definition.

4 Conclusions

In this paper we presented DrawNET++, a prototype system that has been developed to support an OO design process of system models and to provide a graphical front-end to existing performance tools. The current version of the tool together with a complete demonstration example can be downloaded from:

http://www.di.unito.it/~greatspn/DrawNET/

References

1. F.Basile, P. Chiacchio, N. Mazzocca, and V. Vittorini Specification and Modeling of Flexible Manufacturing Systems Using Behaviours and Petri Nets Building Blocks In *Proc. of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE99)*, Los Angeles, USA, 17-18 May 1999.
2. S. Bernardi, S. Donatelli, and A. Horváth. Compositionality in the GreatSPN tool and its use to the modelling of industrial applications. Accepted for publication on *Software Tools for Technology Transfer*.
3. G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *in Performance Evaluation, vol. 24, no.1-2, Nov. 1995*, 1995. 47-68. http://www.di.unito.it/ greatspn/
4. R.M.L.R. Carmo, L.R. de Carvalho, E. de Souza e Silva, M.C. Diniz and R.R. Muntz. Performance/availability modeling with the Tangram-II modeling environment. Performance Evaluation 33 (1998), 45–46.
5. D. Deavours and W. Sanders, Möbius: Framework and Atomic Models. In Proc. 9th International Workshop on Petri Nets and Performance Models, Aachen, Germany, September 2001
6. G. Franceschinis, C. Bertoncello, G. Bruno, G. Lungo Vaschetti and A. Pigozzi SWN models of a contact center: a case study In Proc. 9th International Workshop on Petri Nets and Performance Models, Aachen, Germany, September 2001
7. M. Gribaudo, A. Valente. Framework for Graph-based Formalisms. In *Proceeding of the first International Conference on Software Engineering Applied to Networking and Parallel Distributed Computing 2000, SNPD'00*, pages 233-236 Reims, France, May 2000, ACIS.
8. W.H. Sanders. Integrated frameworks for multi-level and multi-formalism modeling. *Proceedings of the 8th International Workshop on Petri Nets and Performance Models, 1999*, 1999. 2-9.

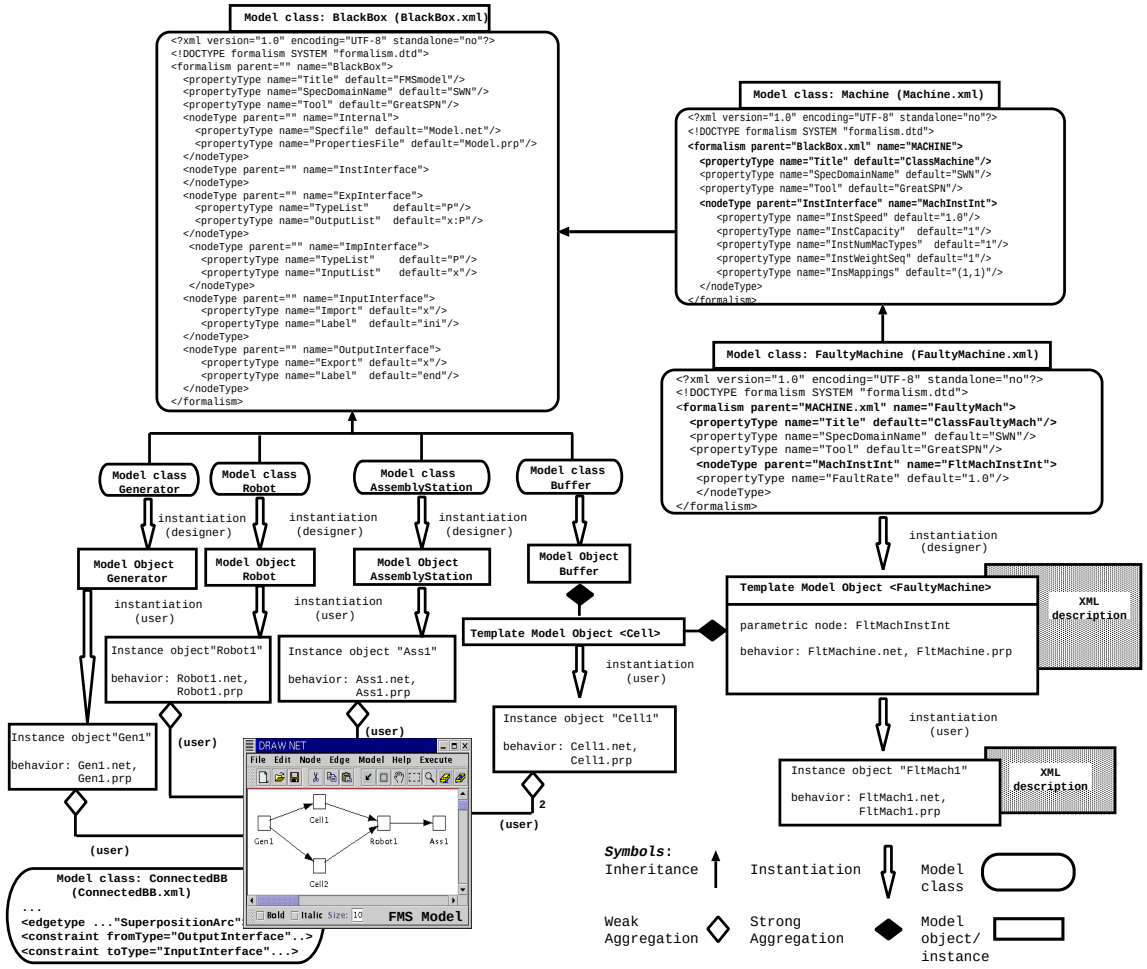

Fig. 1. Construction of a complete model