Petri Nets-Based Automatic Generation GUI Tools for Embedded Systems

Luís Gomes †, *Senior Member, IEEE*, and João Lourenço ‡
† Universidade Nova de Lisboa / UNINOVA, Lisboa, Portugal
‡ Universidade Nova de Lisboa, Lisboa, Portugal
lugo@uninova.pt, joaoplourenco@hotmail.com

***Abstract* — This paper presents a set of development tools for model-based automatic generation of Graphical User Interfaces (GUI) for embedded systems. Two main tools are described: one allowing the definition of the graphical characteristics of the synoptic, named as "Animator", and the second one responsible for the embedded control execution part, named as "Synoptic", and integrating real-time up-dating of the graphical user interface (the synoptic part). In this sense, the embedded system is seen as an ordinary embedded control system integrating the associated graphical user interface, reacting as a SCADA (Supervisory, Control, and Data Acquisition) system (even not supporting the distributed nature of SCADA systems). The behavioral model of the system is specified through a IOPT model (Input-Output Place-Transition Petri nets model), which is represented using a PNML (Petri net Markup Language) notation. An automatic code generator from PNML to C will provide the specific code to be linked with the core code of the graphical user interface. The Petri net model behavior exhibits static and dynamic characteristics, which are associated with the graphical characteristics of the synoptic through a set of dedicated rules. The application of the tools to a simple embedded system for a parking lot control is presented.**

***Keywords* — Design Automation, Graphical User Interface, Model-based development, Petri nets.**

I. Introduction

THE goals of the work presented in this paper are two fold:

- On one hand, usage of the model-based attitude to support the development of embedded systems;
- On the other hand, to establish and automate as much as possible the relationships between the characteristics of the embedded systems behavioral model (used for control execution) and the status of graphical attributes of the associated graphical user interface.

In other words, the goal is to take advantage of the widely accepted model based attitude for development of embedded systems, integrating the automatic generation of the associated graphical user interface with the support of a set of design automation tools. In this sense, the work focuses on the development of embedded control systems integrating the controller part as well as the associated graphical user interface, producing what can be considered as embedded SCADA (Supervisory, Control, and Data Acquisition) systems, where on top of the usual tasks of interacting with operators of the plant associated with the SCADA systems, the embedded system also integrates autonomous control strategies, as usual in ordinary embedded systems.

Several models of computation have been used for embedded systems behavioral modeling.

Among them, Petri nets [1] are often referred as very suitable for the task of embedded system design [2]. Several features supported by Petri nets models are considered very useful for construction of behavioral models for embedded systems, namely:

- A precise syntax and semantics.
- Explicit and readable modeling for parallelism, synchronization, resource allocation and consumption.
- The graphical visualization of models.
- Support for bottom-up (composition) and top-down (refinement) attitudes.
- Possible extensions to time modeling.

The works presented in this paper take advantage of the referred Petri nets characteristics augmented with integration within a set of Petri nets tools under development [3] [4] [5].

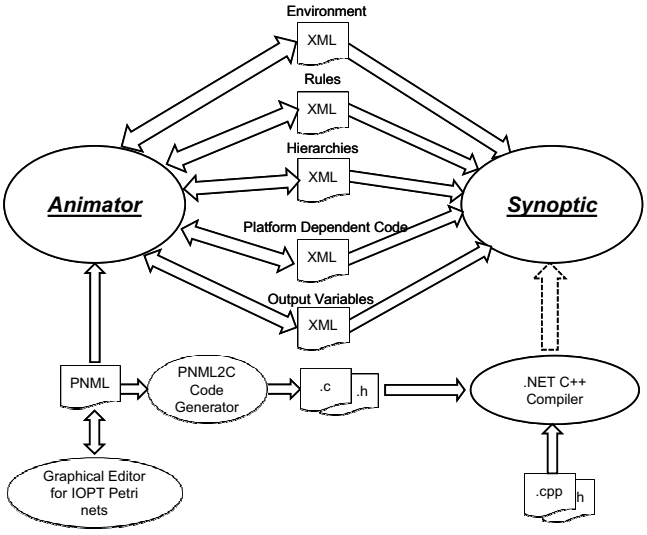

Figure 1 – System architecture.

In order to properly introduce the different aspects of the proposed methodology and tools, Figure 1 presents the main steps of the development flow.

The flow starts at the bottom left of the figure, when a graphical Petri net editor is used in order to produce the behavioral model of the system. The graphical editor is built upon the core code of Snoopy [6], which was kindly provided by the Group of Data Structures and Software Dependability at Brandenburg University of Technology Cottbus in order to support an extension tuned for the edition of IOPT models (Input-Output Place-Transition Petri nets models) [7].

The graphical editor allows the creation of Petri net models in the usual graphical and interactive way, producing PNML representations of the models [8] [9], in order to assure interoperability and integration with other tools (under development on-site and available from other academic groups and companies).

One of the key tools that take advantage of the availability of the PNML representations is the PNML to C code generator tool [3] [4] [5], which can provide ANSI C code amenable to be used in plenty of different platforms and with different goals in mind. As examples of different roles, one can refer that the same generated C code can be used for direct execution of the Petri net model (as in the tools presented in this paper), integrated with verification tools, like state space construction (verification activities are more relevant as the complexity of the system increases), or to support the "token-game" within simulators. Portability assuring its usage within different platforms is also a key feature, from general-purpose platforms (like PCs and workstations), to dedicated microcontrollers (including low-cost microcontrollers), and including reconfigurable platforms (like FPGAs - Field Programmable Gate Arrays), where availability of microprocessors as IP in cooperation with dedicated hardware could improve drastically the performance of the system. This flexibility is supported by a set of specific "platform-dependent code", which is responsible to link the general-purpose ANSI C generated code with the specificities of a physical platform.

One main feature of the proposed methodology for automatic generation of graphical user interfaces integrated with the controller part is the capability to establish relationships between the characteristics of the Petri net model and the characteristics of the graphical features of the user interface (in line with the Model-View-Controller reference pattern for GUI development [10]). This is accomplished by the "Animator" application, left top corner of Figure 1, which is responsible for defining the referred relationship between model and graphical features. In this sense, this application has several outcomes characterizing the execution environment, including the set of rules associating Petri net model characteristics and graphical features ("Rules"), and the definition of the overall graphical layout of the synoptic ("Environment"), all stored as XML files.

Finally, the "Synoptic", referred at the right top corner of Figure 1 is the “embedded SCADA” application that also integrates capabilities for embedded execution of the generated C code. It is the on-line part of the environment that presents the synoptic environment, executing the Petri net model and updating the graphical features of the user interface.

The paper continues in Section II with a brief presentation of the Input-Output Place-Transition (IOPT) Petri net class that is used for the modeling of the embedded system behavior. Section III briefly presents the running example of a parking lot controller, where the set of tools were used, and Sections IV and V present the *Animator* tool and the *Synoptic* application main features. Finally, Section V concludes and refers to some future works (already started).

II. The Input-Output Place-Transition Petri Nets

Petri nets can be seen as a generalization for state machines: transitions can originate more than one active state, thus modeling parallelism, and several states may need to be active to enable a transition, thus modeling synchronization.

Besides, Petri nets are an inherently concurrent model as several enabled transitions can fire concurrently and any number of states can be active at the state time. Petri nets also have a simple graphical representation: passive entities (e.g. states or resource), named places, are modeled as circles or ellipses; active entities (e.g. actions, functions), named transitions, are modeled as rectangles or bars.

Places can only be connected to transitions and transitions can only be connected to places. The connections are specified by directed arcs. Each transition can fire when it is enabled, and it is enabled when all its input arcs are connected to marked places.

Due to its simplicity, numerous Petri net classes have been developed, each one with the convenient ad-hoc characteristics for the systems to be modeled, including modeling for input and output signals dependencies. So, the net becomes non-autonomous, in the sense of the interpreted and synchronized nets of David and Alla [11], and Silva [12].

A. Brief Presentation of the IOPT net class

This section briefly presents the Input-Output Place-Transition Petri nets class (IOPT) (as defined elsewhere [3]). This class is defined as an extension to the place-transition net class (e.g. [1]) allowing the construction of controller models. To that end, this class of nets extends place-transition nets with the specification of input and output signals and events.

The events and signals allow the specification of interactions between the controller model (the net) and the environment, as in interpreted and synchronized nets [11] [12], as in other proposals like [13] [14].

When fired, transitions can change output signals. This is also possible based on place markings. To each transition, it is also possible to associate priorities, guards that use external signals, as well as input and output

events. The IOPT nets have maximal step semantics: whenever a transition is enabled, and the associated external condition is true (the input event and the input signal guard are both true), the transition is fired.

III. Motivating Example

The example to be used to illustrate tools usage is the controller for a parking lot (with a simple layout to allow presentation in a small space). It is important to note that the proposed methodology is applicable to any system, of any complexity, as far as the designer is able to produce the associated model; this means that the methodology scales well. It is also possible to hierarchically structure the synoptic, in order to accommodate monitoring of large plants.

Additional examples could be found at the tool webpage [5], following the link associated with Tools tab, Animator tool.

The parking lot has one entrance and one exit, as presented in Figure 2. The emphasis was put on the identification of the input and output signals coming from the entrance/exit areas to the controller. The behavior of the system can be briefly described as follows: whenever there is a parking place available, a new car can enter the parking lot after activating the *arrive* signal and take a ticket from the machine (that will activate *gotTicket*); the semaphore/barrier will open until the *arrive* signal stays active. Procedures for exiting the park are similar. Figure 3 presents a simplified IOPT model where one can identify three areas: at the left hand side, the entrance area is represented, while at the right hand side the sub-model for the exit area is presented; in the middle, two places (*CarInsideZone* and *FreePlaces*) model the parking lot occupancy. At the bottom, signals and events used in the model are identified. Some modeling variants were used to illustrate the flexibility of the formalism. For the entrance area, events were used (namely *arriveOut*, *arriveIn*, and *identified*), while for the exit area, conditions on signals were selected (namely *leave==1*, *leave==0*, and *pay==1*, which are using a C language sintax). In this sense, it is up to the modeler to choose the type of model that he/she wants to produce, emphasizing event dependency or signal dependency.

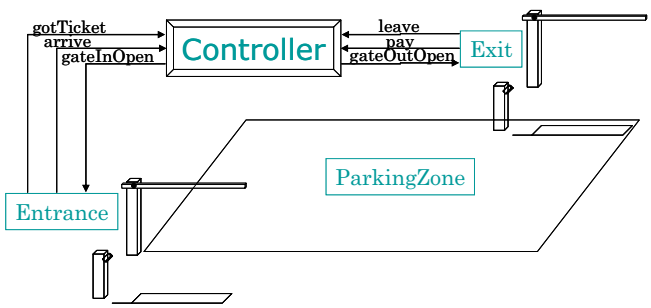

Figure 2 - Parking lot layout and controller interconnection.

Overall, the example is a kind of producer-consumer system, which is widely used for pedagogical purposes in different areas of sciences and engineering. However, in the current work, emphasis is put on the modeling of the controller behavior, where connections to the real world are foreseen, namely input and output signals.

IV. The Animator Environment

The main objective of the *Animator* tool is to prepare a set of files to be used as automatic configuration for the *Synoptic* application in run-time. These files include the specific environment characteristics and characterization of the rules associating the Petri net model and the GUI characteristics. First, let's show how these files can generate and create adequate models to support the construction of that environment.

A. PNML Generation

Initially the user needs to build the behavioral model of the system using a tool for graphical editing of IOPT Petri nets [7]. The tool can generate the associated PNML representation. This tool, based on the core code of Snoopy environment [6], as referred, besides generating the PNML representation, has the functionality to add or edit additional features of the IOPT class, namely input or output events and signals. The resulting file, which is represented in PNML (XML-based interchange format for Petri nets) contains all the characteristics of the IOPT Petri net. A very small fragment of the listing of the generated PNML file for the presented example associated with the representation of the *identified* event, which is associated with the signal *gotTicket*, is shown below (the event *identified* will be generated at the rising edge of the signal *gotTicket*).

```
<?xml version="1.0" encoding="iso-8859-1"
                                stand-alone="no" ?>
<Snoopy revision="1.4-IOPN" version="2">
  <pnml>
    <net id="1" name="Car Parking" type="IOPT">
      <input>
        <event edge="up" id="identified"
                  level="0" signal="gotTicket">
          <graphics>
            <position page="1" x="443"
                                y="320"/>
          </graphics>
        </event>
 (...)
```

B. From Use Cases to models

It is important to stress that before being able to produce a sound behavioral model for the controller of the system, the modeler can take advantage of UML Use Cases in order to identify the functionalities and requirements of the system. To produce the system model starting from the use cases description, the following steps are foreseen, according with Figure 4:

- Translation of each use case into a formal (partial) model;
- Translation of each one of the partial models into a behavioral equivalent Petri net model;
- Merge all partial models into the whole Petri net system model.

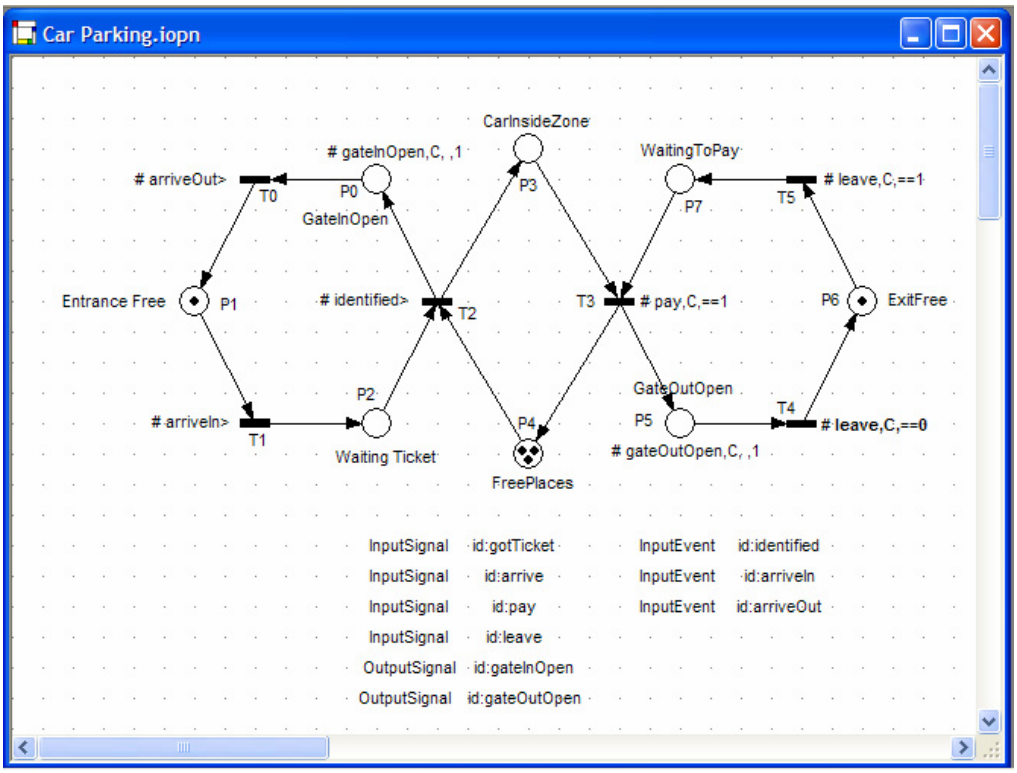

Figure 3 - Parking lot IOPT Petri net model.

The first step is accomplished by the designer through requirement's inspection of each use case and manually produce associated model.

The second step is a translation of each of the partial models into a behavioral equivalent Petri net model. In this sense, the Petri net model acts as an intermediary representation that will be supported by a tool set.

Finally, the merging of all partial models into the whole system model, which is the main goal of this phase allowing construction of the system model, can be easily accomplished through the use of the net addition operation, defined elsewhere [15], and associated tool OPNML2PNML [16], also available at [5].

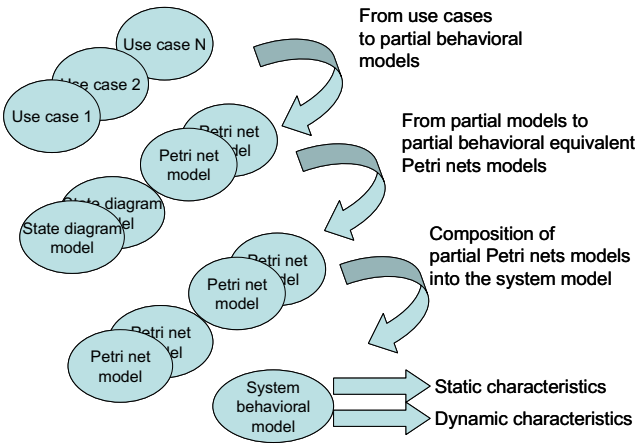

Figure 4 - From use cases to overall system behavioral model.

A. *From models to graphical representation*

The next goal after constructing the model is to create one or various synoptic layouts by adding background images and animated images into the respective child windows. Every child window can be resized (background image will resize automatically), and later can become independent from the main window. This functionality allows seeing different parts of the plant/system under control (associated with different parts of the Petri net model) in different windows/monitors. These windows could be arranged hierarchically. Different child windows can be merged, by joining the backgrounds.

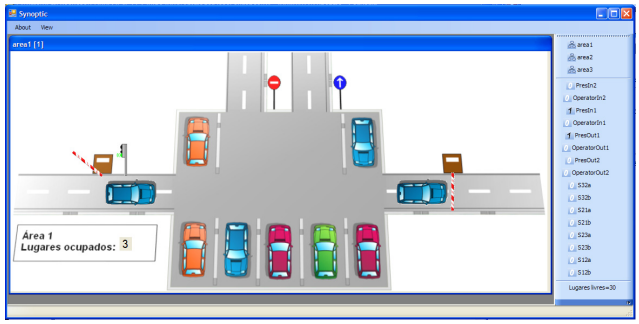

Figure 5 – Generated synoptic for the parking lot example.

Several characteristics commonly used in SCADA GUI are available for use. For instance, one additional available functionality allows animated images (superimposed to the background). This animated images can be resized and moved into different locations, and can also be chosen if the user wants transparency around the image or not, so the pictures can have arbitrary shapes (not only squares and rectangles).

After composing the environment, the whole synoptic environment can be saved as an XML file, that afterwards can be opened by the *Synoptic* application described in the next section. A snapshot of the generated synoptic for the parking lot example is presented in Figure 5.

B. *Rules for the animated images*

The *Animator* tool has a dedicated editor that is used to create a set of rules to allow the association between specific IOPT Petri net model characteristics and specific features of the graphical layout used by the synoptic. These rules will describe the behavior of an animated or static image in the synoptic when a certain trigger condition occurs in the IOPT Petri net model. That trigger condition could be dependent on places, transitions, events or signals of the Petri net model. So the graphical user interface, which means the whole synoptic, will be automatically updated on the occurrence of an event produced as a change at the Petri net model state. For that end, the information presented in the PNML file is loaded to the *Animator* tool, in order to allow the designer to define the desired set of rules.

Each rule has a simple structure, according with format *IF antecedents THEN consequents*, where *antecedents* are dependent on the Petri net model characteristics and *consequents* identify graphical effects.

As shown in Figure 6, static and dynamic model characteristics of the Petri net model can be considered to be included in each rule. The static characteristics are related with the marking of the Petri net model and status of the input and output signals, while the dynamic characteristics concerns with the firing of the transition and occurrence of events.

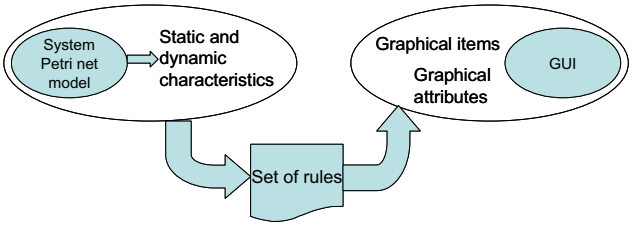

Figure 6 - Relation between model characteristics and graphical attributes.

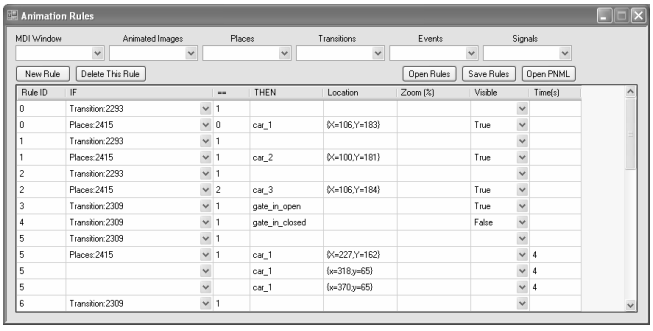

Figure 7 - Animation rule editor interface.

Taking this into account, it is possible to create the set of rules for a specific control application.

An example on how the rules can be constructed is shown in Figure 7 (that presents a snapshot of the rule's editor interface). The rule is stored in a XML format, as the one presenting next:

```
<?xml version="1.0" encoding="utf-8"
                              stand-alone="yes" ?>
<!-- Rules for the animated images--> <Rules>
 <Rule>
   <RuleID>0</RuleID>
   <if>Transition:2293</if>
   <equals>1</equals>
   <and>Places:2415</and>
   <equals>0</equals>
   <then>car\_1</then>
   <location>{X=106,Y=183}</location>
   <visible>True</visible>
 </Rule>
 <Rule>
(...)
```

The operators that are supported for construction of conditions and composition of antecedent part are the relational operators: == (equals), ! (different), >= (more or equal), <= (less or equal), > (more), and < (less).

Summarizing, rules can be used to:

- Replace an image by another image (in order to reflect the current status of a resource, for instance);
- Move an image to other location immediately or under a timer control (to simulate flows at the synoptic);
- Move images following a specific path;
- Show or hide an image;
- Resize an image.

V. Synoptic Application

As presented in Figure 1, after getting the generated PNML file of the Petri net model, a PNML to C automatic code generator tool is used (the tool is also available through [5]). This tool is used to produce the Petri net implementation code, written in ANSI C language, and used to execute the model. The generated C files, and his respective headers are linked with the (previously prepared) *Synoptic* project files, in order to build the executable file of the *Synoptic* application.

The *Synoptic* application is able to automatically read the information provided by the *Animator* tool, and configure itself for operation. The configuration information includes several files, namely environment definition, set of rules associating the Petri net model with graphical characteristics, information on hierarchical structuring of the windows of the synoptic, and information associated with input/output signals and events and how to get/put them from/into the physical environment.

The *Synoptic* is able to read the described configuration data, execute the Petri net model on a cycled basis, checking changes and determining the next state and automatically update the GUI according with the pre-defined rules.

VI. Conclusion

The paper presents two applications supporting the automatic generation flow for graphical user interfaces

associated with embedded systems, working as centralized SCADA systems. The first application, the *Animator*, is used to define the synoptic characteristics, while the second application, the *Synoptic*, is able to execute a behavioral Petri net model of the controller and continuously updating the graphical user interface, being able to react to input changes, coming from the user or from the process under control.

An executor of the Petri net model, which is initially represented using PNML and translated into C code using a tool for automatic code generation from PNML to C, is linked with the Synoptic project files allowing automatic generation of the SCADA application.

The whole set of tools is publicly available at the FORDESIGN website [5], by the date of its complexion (April 2008).

As future work, in order to take advantage of the already available tools, two main lines of research and developments are foreseen.

On one hand, it is foreseen to take advantage of a PNML to VHDL code generator (also under development within the FORDESIGN project) and exploit the usage of the proposed methodology in order to have the whole embedded control system in an FPGA or in a SoC (System-on-a-Chip) (integrating associated graphical user interface).

On the other hand, it is foreseen to extend the support to other modeling formalisms for specification of the embedded system behavior, other than IOPT, namely other classes of Petri nets, like Colored Petri nets [17], StateCharts [18], and Sequence Diagrams, just to mention the most relevant for the foreseen works.

ACKNOWLEDGMENTS

The authors acknowledge the collaboration received from other members of the FORDESIGN project. A special reference goes to Ricardo Nunes (Snoopy-IOPT Petri net editor developer) and Tiago Rodrigues (PNML2C tool developer).

REFERENCES

[1] Wolfgang Reisig. Petri nets: an Introduction. 1985. ISBN 0-387-13723-8. Springer-Verlag New York, Inc.

[2] Claude Girault and Rudiger Valk. Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications. Springer. ISBN 3-540-41217-4. 2003.

[3] Luís Gomes, João Paulo Barros, Anikó Costa, and Ricardo Nunes. The Input-Output Place-Transition Petri Net Class and Associated Tools. INDIN'2007 - 5th IEEE International Conference on Industrial Informatics, 23-26 Julho 2007, Vienna, Austria.

[4] Luís Gomes, João Paulo Barros, and Anikó Costa. Petri Nets Tools and Embedded Systems Design. PNSE'07 - International Workshop on Petri Nets and Software Engineering, Siedlce, Poland, June 25-26, 2007

[5] FORDESIGN project website. http://www.uninova.pt/fordesign

[6] Data Structures and Software Dependability Brandenburg University of Technology Cottbus. SNOOPY's home page. 2007. http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html.

[7] Ricardo Nunes, Luís Gomes and João Paulo Barros. A Graphical Editor for the Input-Output Place-Transition Petri Net Class. ETFA'2007 - 12th IEEE Conference on Emerging Technologies and Factory Automation, September 25-28, 2007; Patras, Greece.

[8] PNML, Petri Net Markup Language. 2004. http://www.informatik.hu-berlin.de/top/pnml/about.html.

[9] Jonathan Billington, Soren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net MarkupLanguage: Concepts, Technology, and Tools. Proceeding of the 24th International Conference on Application and Theory of Petri Nets. W. van der Aalst and E. Best (eds.). 483--505. 2003. LNCS-2679.

[10] Glenn E. Krasner and Stephen T. Pope, A cookbook for using the model-view-controller user interface paradigm in Smaltalk-80, Journal of Object-Oriented Programming, 1(3), pp. 26-49, August/September 1988

[11] R. David and H. Alla. Petri Nets \& Grafcet; Tools for Modelling Discrete Event Systems. Prentice Hall International (UK) Ltd. 1992.

[12] Manuel Silva. Las Redes de Petri: en la Automatica y la Informatica. Editorial AC, Madrid, 1985.

[13] M. Rausch and H.-M. Hanisch. Net Condition/Event Systems with Multiple Condition Outputs. ETFA'95, Paris, Oct. 1995. Proceedings, Vol. 1, pp. 592-600.

[14] P.H. Starke, H.-M. Hanisch. Analysis of signal/event nets. ETFA '97 - 6th International Conference on Emerging Technologies and Factory Automation Proceedings; Los Angeles, CA, USA; 9-12 Sep 1997. ISBN 0-7803-4192-9. pp. 253-257.

[15] João Paulo Barros, Luís Gomes; Net Model Composition and Modification by Net Operations: a Pragmatic Approach. INDIN'2004 – 2nd IEEE International Conference on Industrial Informatics; 24-26 June 2004; Berlin, Germany.

[16] João Paulo Barros, Luís Gomes. Operational PNML: Towards a PNML Support for Model Construction and Modification. In Workshop on the Definition, Implementation and Application of a Standard Interchange Format for Petri Nets; Satellite event at the International Conference on Application and Theory of Petri Nets 2004, Jun 2004, Bologna, Italy

[17] Kurt Jensen. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use - Volume 1 Basic Concepts. Monographs in Theoretical Computer Science. Springer-Verlag. Berlin, Germany. 1997. ISBN: 3-540-60943-1.

[18] David Harel. Statecharts: A Visual Formalism for Complex Systems. Science of computer Programming. 1987. vol. 8. 231-274.