FROM PETRI NETS TO EXECUTABLE SYSTEMS: AN ENVIRONMENT FOR CODE GENERATION AND ANALYSIS

João Paulo Barros[1,2*], Luís Gomes[1], Rui Pais[1,2], and Rui Dias[1]

[1]*Universidade Nova de Lisboa / UNINOVA, Portugal*
[2]*Instituto Politécnico de Beja, Escola Superior de Tecnologia e Gestão, Portugal*

Keywords: Petri nets, modularisation, code generation, executable models, verification, domain specific languages.

Abstract: There is an increased awareness regarding the importance of executable system's specifications, in particular, graphical specifications. Although most Petri nets variants are recognised as a versatile formalism, with an intuitive graphical specifications and a precise semantics, most Petri nets tools limit themselves to graphical editing and some type of simulation, system analysis, or both. This paper presents a new development environment based on Petri nets. This environment enables the use of *ad-hoc* Petri net classes as domain specific languages and allows the net models compositions and evolution through a set of orthogonal and generic modification operations. It also generates ANSI C code (easily extendable to other executable code) amenable to be implemented in general-purpose hardware platforms, without sophisticated resources available. Additionally, one major environment feature is the use of the same generated executable code, both for simulation and for analysis purposes.

1 INTRODUCTION

Already in 1992, a paper by David Harel (Harel, 1992), proclaimed the advantages of model executability (simulation) and code generation, as a way to fight system complexity. And, especially since OMG's MDA initiative (OMG, 2003a), the importance of modelling environments allowing model execution and code generation is widely acknowledged (OMG, 2003b).

In spite of this increased awareness regarding the importance of executable system specifications and, in particular, graphical specifications, most Petri nets tools limit themselves to graphical editing, some type of simulation, system analysis, or both. Code generation is much more difficult to find. In the reference Petri net tools database (Petri Nets Tool Database, 2004), from a total of 51 Petri nets tools, only 3 mention code generation: the SIPN-Editor, a programming tool for PLC programmers, generates IL programs; the CPN/Tools mentions code generation but only for simulation purposes; the SYROCO tool generates C++ code from a very high-level Petri net based language supporting very significant extensions to Petri nets, e.g., inheritance, and dynamic instantiation. This paper presents a development environment for system development based on Petri nets. It can be defined as an environment for system development based on executable, and implementable, generic Petri net models, with strong support for model structuring and model modification. By generic Petri net models we mean that the support for new Petri net syntax or semantics can easily be added to the environment.

The paper makes a brief presentation of the proposed development environment, namely its architecture, its behaviour, and its two main system tools: the *PnEditor* and the *PnGenerator*. Finally, the paper summarises the system's innovations.

2 THE DEVELOPMENT ENVIRONMENT

The development environment is based on two tools:

1. A graphical editor (named *PnEditor*) supporting any Petri net model defined in the Petri net Markup Language (PNML) (Jüngel et al., 2000; Billington et al., 2003)[2]. It also supports hierarchical structuring and a set of modification operations.

[*]Work partially supported by a PRODEP III grant (Concurso 2/5.3/ PRODEP/2001, ref. 188.011/01).

[2]As all our models will correspond to some Petri net type, the used format is the PNML. The PNML is an emerging standard for a Petri net interchange format. It is based

Paulo Barros J., Gomes L., Pais R. and Dias R. (2004).
FROM PETRI NETS TO EXECUTABLE SYSTEMS: AN ENVIRONMENT FOR CODE GENERATION AND ANALYSIS.
In *Proceedings of the First International Conference on Informatics in Control, Automation and Robotics*, pages 464-467
DOI: 10.5220/0001142104640467

2. A code generator tool (named *PnGenerator*) that is able to generate ANSI C code (and is easily extendable to other languages) for net execution. The generated code is amenable to be executed in hardware with limited resources. Besides, the same generated code can be used to simulation and verification purposes.

A high-level view of the environment, explained along this section, is presented in Fig. 1.

Typically, the user interacts with the PnEditor. The editor allows the user to generate Petri net models based on a previously read Petri net type definition (PNTD). This is a XML file defining the Petri net type to be used. More specifically, it defines the set of textual annotations than can be associated to each net element. The editor configures itself based on the read PNTD: only the net elements defined in the PNTD are made available in the graphical editing commands.

The editor is also able to generate a non-hierarchical specification from a hierarchical model being edited. We call this version, the *flat model*. This means that we view the hierarchical model as a graphical convenience. On one hand, this has the advantages of a modular specification, namely readability, modularity, and reusability. On the other hand, the reduction to a flat model guarantees that the generated Petri net can easily remain closer to well-studied Petri nets for which numerous verification possibilities exist (e.g. elementary net systems (Rozenberg, 1987), free-choice nets (Desel and Esparza, 1995), and Place/Transition nets (Desel and Reisig, 1998)).

The PnGenerator reads flat net specifications and generates ANSI C code capable of executing the model (the *executable model*). It can also generate interface code that is linked to the executable code. This allows the use of the same executable code also for simulation and analysis tasks.

The simulation is, typically, carried out by the interaction of the PnEditor and PnGenerator tools:

1. the PnEditor sends (by file) upon user request, an initial flat Petri net;
2. the PnGenerator identifies the possible next steps and returns that information to the PnEditor in the form of several possible evolutions for the given flat net.
3. the user chooses one of the possible execution steps
4. the PnEditor asks the PnGenerator to executes the specified step
5. the PnGenerator returns the modified model elements to the PnEditor
6. the PnEditor updates the model

on XML and is part of the ongoing effort for a High-level Petri net ISO standard (Petri nets Standard, 2004).

Additionally, given a flat model, the PnGenerator is able to generate automatic simulations (without user intervention) logging them in text files.

Finally, given a flat model, the PnGenerator is also able to generate the respective state graph and to use it for analysis purposes, namely, deadlock detection and reachability analysis.

The environment architecture also makes evident a clear independence between modelling and code generation tasks. The creation of graphical models and the definition of Petri net types can, thus, easily evolve independently from the code generation and simulation tasks. This makes the environment more open as it will allow both tools to be used outside of the initial foreseen close cooperation: other tools can totally, or partially, replace or complement, the existent ones.

2.1 The PnEditor

The PnEditor is a typical Petri net graphical editor with three main additional characteristics: (1) the use of the emergent standard for Petri net model interchange; (2) the support for model modification operations; (3) the interaction with the PnGenerator for simulation and verification tasks. Additionally, the PnEditor is able to generate non-hierarchical (flat) models from hierarchical ones.

The PnEditor relies on the Eclipse open source project (Eclipse, 2004) mainly due to the availability of a suitable framework, named Graphical Editing Framework (GEF), for the development of graphical editors (GEF, 2004).

The PnEditor also allows the specification of net modifications through the use of textual expressions. These textual expressions correspond to a simple algebra around a set of operations on net instances, places, and transitions. Net instances are elements of *net vectors*. Given a net model N, a net vector of net models N is a list of n instances of N nets, denoted $N[1 \ldots n]$ where each net instance is denoted $N[n]$. All net components of a net instance $N[i]$ have their denotations suffixed by $N[i]$. For example, place p_1 in a net instance $S[2]$, becomes $S[2].p_1$. This allows a convenient way to refer to several distinct node instances of a given net (Gomes et al., 2002; Gomes and Barros, 2003; Barros and Gomes, 2004a).

The net instances and the respective places and transitions, can be modified by a small set of operations defined elsewhere (Barros and Gomes, 2004a), namely *net addition* and *net subtraction*. The net modification operations can be applied at any time some model modification is needed. Yet, net addition can also be used in a structured way as the underlying support for the hierarchical structuring of the net model (Gomes and Barros, 2003) and for a particular type of transition fusion, named *synchronous group*, useful for object-oriented design (Barros and Gomes,

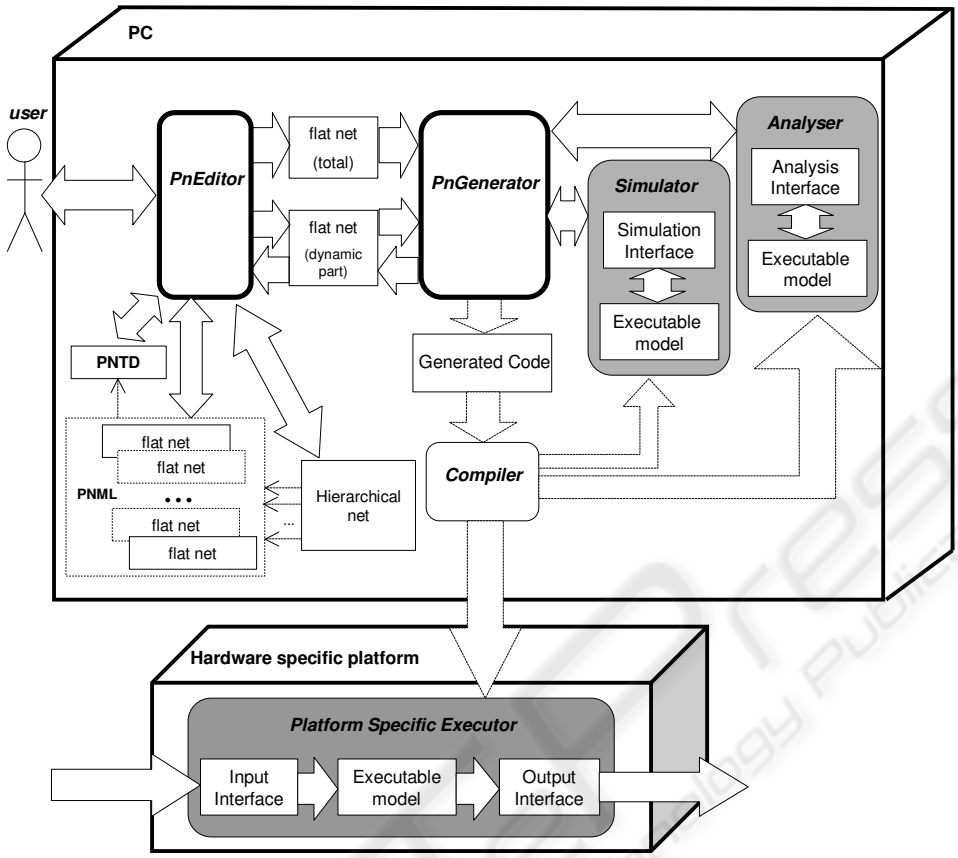

Figure 1: The development environment architecture.

2004b). Net subtraction allows the undo of a previous net addition operation.

2.2 The PnGenerator

Generically, we can view the PnGenerator tool as allowing the developer to execute compiled Petri nets models. This execution has three variants depending on the use we want to give to the model execution. Those variants correspond to the following functionalities:

Code generation The Petri net model is read from a text file (in PNML) format and executable code (e.g. ANSI C) is generated. When compiled, this code, is able to execute the Petri net.

Simulation This corresponds to the previously described interaction between the PnEditor (the user) and the PnGenerator.

Verification The verification uses the same code as the simulation in order to produce the associated state space. It is sufficient to generate additional code stubs allowing the interface between the executable code and the user.

Compared to the usual interpreted model execution, the model compilation allows smaller footprints and better performance for the executable code. This results from the significant optimisations that can be done at compile time: for example, if the net class allows priorities in transitions, but they are not used in the model, then the generated code will not even contain code testing their occurrence or value. And the same is true for other similar cases.

Just like all model execution related knowledge, the Petri net class semantics is hard-coded in the PnGenerator. Basically, for each new Petri net class to be supported, one new class needs to be added to each of three packages: (1) ”Specific Petri nets classes Read”, to read the specific data in the PNML file; (2) ”Generate Specific Petri net Static Code” to specify the generation of code for the Petri net structure in the desired output executable language (e.g. ANSI C); (3) ”Generate Specific Petri net Dynamic Code” to specify the generation of code for the Petri net execution in the desired output executable language. Note that distinct dynamic behaviours can be specified for a given Petri net class.

3 CONCLUSIONS AND FUTURE WORK

Here, we highlight the main characteristics which, when taken together, make the presented environment innovative:

Handling of multiple Petri nets formats The PnEditor is able to configure itself accordingly to a given Petri net type (defined accordingly to the PNML emergent standard). In this way the editor can be used to create models given in distinct Petri nets based languages, with varying degrees of platform independence. In summary, the PnEditor is able to define and use multiple Petri nets as domain specific languages.

Model structuring As the support is independent from the respective net class, the PnEditor allows hierarchical structuring of any Petri net model. Additionally, a special type of transition fusion enables the use of High-level Petri nets for Object-oriented design.

Flexible model modifications The definition of a set of textually specifiable model modification operations (represented by algebraic expressions) complements the graphical nature of the Petri nets with a compact notations for the specification of very large models. This also allows higher levels of modularisation.

Versatile Model Compilation The environment supports the compilation of the designed models and the use of the resulting executable code not only for simulation purposes but also for verification and final implementation in a suitable hardware platform.

Generation of Platform Specific code The PnGenerator generates ANSI C code and the ad-hoc code stubs allowing the execution of the models in specific hardware or software platforms.

The environment was tested using elementary Petri nets for a manufacturing system model and using Place/Transition nets for a parking lot access controller. Finally, code generation will be extended to include hardware description languages, namely VHDL.

REFERENCES

Barros, J. and Gomes, L. (2004a). Net model composition and modification by net operations: a pragmatic approach. In *Proceedings of the 2th IEEE International Conference on Industrial Informatics (INDIN'04)*. (to appear).

Barros, J. and Gomes, L. (2004b). On the use of coloured Petri nets for object oriented design. In Michelis, G., D. and Diaz, M., editors, *Lecture Notes in Computer Science; , 25th International Conference on Application and Theory of Petri Nets 2004*. Springer. (to appear).

Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., and Weber, M. (2003). The Petri net markup language: Concepts, technology, and tools. In van der Aalst, W. and Best, E., editors, *Proceeding of the 24th International Conference on Application and Theory of Petri Nets*, volume 2679 of *LNCS*, pages 483–505, Eindhoven, Holland. Springer-Verlag.

Desel, J. and Esparza, J. (1995). *Free choice Petri nets*. Cambridge University Press.

Desel, J. and Reisig, W. (1998). Place/transition Petri nets. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:122–173.

Eclipse (2004). Eclipse.org. `http://www.eclipse.org`.

GEF (2004). Graphical editing framework. `http://www.eclipse.org/gef/`.

Gomes, L. and Barros, J. (2003). On structuring mechanisms for Petri nets based system design. In *Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, pages 431–438. IEEE Catalog Number: 03TH8696.

Gomes, L., Barros, J., and Costa, A. (2002). Petri net model node structuring techniques for embedded system design. In *Proceedings of the 5th Portuguese Conference on Automatic Control (CONTROLO'2002)*, Aveiro, Portugal. Associação Portuguesa de Controlo Automatico (APCA).

Harel, D. (1992). Biting the silver bullet: Toward a brighter future for system development. *Computer*, 25(1):8–20.

Jüngel, M., Kindler, E., and Weber, M. (2000). The Petri net markup language. In Phillipi, S., editor, *Workshop Algorithmen und Werkzeuge fr Petrinetze*.

OMG (1997-2003a). OMG Model Driven Architecture. `http://www.omg.org/mda`.

OMG (1997-2003b). Presentations and papers. `http://www.omg.org/mda/presentations.htm`.

Petri nets Standard (2004). Petri nets standard. `http://www.daimi.au.dk/PetriNets/standardisation/`.

Petri Nets Tool Database (2004). Petri nets tool database. `http://www.daimi.au.dk/PetriNets/tools/db.html`.

Rozenberg, G. (1987). Behaviour of elementary net systems. In *Advances in Petri nets 1986, part I on Petri nets: central models and their properties*, pages 60–94. Springer-Verlag.