Cloud Based IOPT Petri Net Simulator to Test and Debug Embedded System Controllers

Fernando Pereira[1,2,3(✉)] and Luis Gomes[1,3]

[1] Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia, Lisboa, Portugal
fjp@deea.isel.ipl.pt, lugo@fct.unl.pt
[2] ISEL, Instituto Superior de Engenharia de Lisboa, Lisboa, Portugal
[3] UNINOVA – CTS, Lisboa, Portugal

Abstract. IOPT-Tools is a cloud based integrated development environment to the design of embedded system controllers and other digital systems, employing the IOPT Petri net modeling formalism. The tools include a graphical editor, a state-space based model-checking subsystem and automatic code generators to deploy the controllers on the target hardware platforms. This paper presents a new Simulator tool that offers the capability to execute embedded system controllers based on IOPT models in a Web browser. To allow the test and debug of embedded system controllers, the Simulator provides options to manipulate the value of input signals, step by step execution, and continuous execution with programmed step frequency and breakpoint definition. Simulation history is recorded, continuously storing information about the entire system state, to enable playback and history navigation. History data can later be exported in spreadsheet format for analysis with external tools and waveform drawing. The tool can be accessed from http://gres.uninova.pt.

Keywords: Embedded systems · Cloud based tools · Petri nets

1 Introduction

The main goal of the IOPT-Tools integrated development environment [1] is to provide a rapid application development framework to the design of embedded system controllers and general purpose digital systems. In such framework, debug and simulation tools assume special importance, allowing the detection of mistakes during the early development phases, before reaching the prototype implementation, greatly contributing to reduce development time and cost and minimize the risk of prototype hardware damage due to controller design errors.

The tools offer a complete Cloud based tool-chain, including a graphical editor, a model-checking sub-system based on state-space computation, automatic code generation tools to produce software code or hardware descriptions, and the simulation and debug tool presented in this paper. All tools have a Web based user interface, with data storage and computing intensive operations executed on the Cloud, in the IOPT-Tools servers.

In this environment, embedded system controllers are modeled using IOPT nets [2], a subclass of Petri nets [3] specifically created for this purpose, with the addition

© IFIP International Federation for Information Processing 2015
L.M. Camarinha-Matos et al. (Eds.): DoCEIS 2015, IFIP AICT 450, pp. 165–175, 2015.
DOI: 10.1007/978-3-319-16766-4_18

of the concepts of input and output signals and events to the traditional Petri net place and transition nodes. This way, controller behavior is expressed using the standard Petri net concepts, but the interface between the controllers, the controlled systems and the user interface is defined using input and output signals and events.

Figure 1 presents the typical IOPT-Tools development work-flow, including all development stages. The first steps, starting with the model edition, debug and simulation, model-checking and property verification are performed inside the tools using a pure software solution. This way, designers can test and debug controllers, focusing exclusively on the controller behavior without distractions from hardware details. The final development stages are also assisted by the automatic code generation tools, minimizing the risk of low level coding mistakes and simplifying the migration to different hardware platforms.

The simulator is usually employed in the second development stage, immediately after a model is designed or suffers changes, the typical use case situations must be simulated to check if the model exhibits the expected behavior. After successful completion of the typical use case sequences, development can progress to the next stages, including model checking and property verification, automatic code generation and prototype implementation.

To better analyze the debug and simulation results, a simulateion history table is automatically recorded, containing all system data for every execution step, including the net marking, input and output signals and events and internal variables. The history can be visualized as a spreadsheet table, exported to create waveform drawings or used to replay a graphical animations of the stored execution steps. The user can also navigate through the history and restart the simulation from any saved history step, to test the system behavior under different input sequences.

The tools can be used from the IOPT-Tools web service, located at http://gres.uninova.pt, and do not require any software installation on the user's personal computer or tablet computer, running directly on the Web browser. New users can log-in using a guest account to experiment the tools, but can also create personal user accounts to store private models.

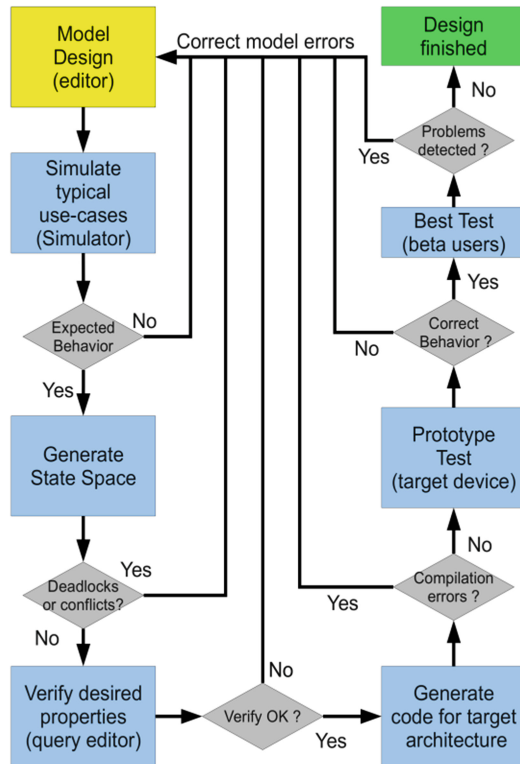

Fig. 1. The typical IOPT-tools workflow

For more information about the tools and IOPT nets, a detailed user manual is also available online at the tools Web site [4].

2 Relationship to Cloud-Based Solutions

The tools presented in this paper are a cloud-based solution. The simulator and the other tools run directly on the user's Web browser, using AJAX (Asynchronous Java-script and XML) principles [5]. All data files are stored on the cloud, in the IOPT-Tools servers, and all tasks requiring heavy duty computation, as state-space calculation and model-checking are also performed on the server, with a Web interface to display the results. This solution minimizes the computation requirements on the user's terminal equipment and can be used from low end personal computers, tablet computers or even smart-phones, as long as a W3C compliant Web browser is available.

On another way, the simulator is also a building block of a distributed architecture proposed in [6], intended to simplify the development of cloud-enabled embedded devices. This architecture minimizes the hardware resources required to implement embedded devices, greatly reducing costs, by storing all user interface data and code on the cloud and executing the user interface logic in the user's Web browser. The embedded device only needs to provide a minimal TCP/IP infrastructure to exchange data with the user interface code running on the browser, including information about the current system state and input and output signals, and receive feedback from the user interface logic.

3 Related Work

The simulator presented in this paper is integrated in a larger tool-chain, including a model editor [7], a model-checking infrastructure composed by a state-space generator and a query system [8] and automatic code generation tools [9]. Additional information about these tools has been published in [1] and a user manual in [4]. The underlying modeling formalism on which the tools are based, IOPT nets, has also been discussed in [2]. A direct connection to the HIPPO system [10] provides additional tools, including the calculation of incidence matrices and analysis of place invariants, concurrency relations and sequential relations.

Computer simulation is widely disseminated over most engineering and scientific fields, and simulators can be found in many development environments [11]. General purpose simulation software applications like Math-lab and Simulink [12] are often used to simulate diverse physical systems, covering a wide range of scientific fields. Among electrical and electronics engineers, simulators like SPICE are very popular for electrical circuit analysis. Like the tools presented in this paper, Matlab and Simulink are often used on the development of embedded system controllers, including automatic code generation tools to deploy models on hardware prototypes. IOPT nets are often used as a high level modeling formalism to the design of digital system circuits.

Simulators are usually employed to study the behavior of systems, in situations where it is not practical, costly or convenient to perform the corresponding experiments in the real world, like the effects of natural disasters, weather prediction, the behavior of sub-atomic particles, or the evolution of human demographics.

Traditionally, Petri nets have been used as a modeling language to run simulations. As the standard Petri net classes are autonomous and lack the capabilities to communicate with the external world, the purpose of these classes are the simulation by itself, and almost all Petri net tool-chains include a Simulator tool. For example, simulators can be found in CPN-Tools [13], CPN-AMI [14] and Renew [15].

Although existing simulators implemented in Java could be executed from Web browsers, the new simulator tool runs entirely inside the Web browser, without requiring any additional virtual machine. This way, simulator usage is not limited to PCs and can be executed on a wide range of hardware platforms and operating systems, from PCs and tablet computers to smart-phones, as long as a W3C compliant Web browser is available.

A previous Animator tool [16], could also be used to run animated simulations of Petri net models, but required the definition of a series of user interface «screens» containing graphic objects to visualize the system state and interact with the user. However, the tool and the simulations produced by this tool do not offer a Web interface and cannot be integrated in the IOPT-tools environment. The development of cloud-enabled version of the Animator tool is planned for the future and the new simulator tool already contains infrastructure code to interact with animated user interface windows build using the future Animator.

4 Research Innovation

The simulator presented in this paper should not be understood as a Petri net simulator, but as an embedded system controller simulator that uses IOPT Petri nets as the underlying modeling formalist. Due to the non-autonomous nature of IOPT nets, the new Simulator tool exhibits fundamental differences from the existing Petri net simulators for autonomous Petri net classes.

With traditional Petri net simulators, the user interaction with the simulator is limited to the definition of an initial marking before starting the simulation and during execution, the user can only choose the transitions that fire on each execution step, from a set of enabled transitions. In contrast, using the new embedded system controller simulator, the user interacts with the simulator by setting the value of input signals and autonomous input events. All enabled transitions will automatically fire in the next execution step, as IOPT nets use a maximal step semantics. In this regard, the IOPT simulator has more similarities with the tools found in the Ladder and Grafcet [17] programmable logic controller tool-chains for industrial automation, than with the autonomous Petri net simulators.

As the user does not have to manually choose which transitions will fire, the Simulator is typically used in continuous run mode, as opposed to the traditional Petri net simulators that are often used in step by step execution mode. As a consequence, the

concept of Breakpoints, popular in software development tools, has been introduced to the new Simulator, enabling the automatic interruption of continuous execution when selected transitions are fired.

As embedded system simulations typically run at very fast step rates, the user may not be able to notice all execution details in real time. To solve this problem, a simulation history mechanism has been added to enable the posterior analysis of the results and export data to draw waveform graphics. To help the user track the system evolution in real time, special attention was given to graphical feedback, dynamically changing the colors of input and output signals, triggered events, marked places, enabled transitions and transitions with guards ready.

From an implementation point of view, the new simulator also differs from traditional Petri net simulators, as it employs a compilation execution strategy instead of using a model interpreter: each time the simulator is called, the Javascript code responsible for implementing the model behavior is rewritten, according to the model in use. This strategy offers improved simulation performance, enabling simulations running at very fast step rates, comparable to the speeds employed in real embedded devices based in micro-controllers.

Finally, to reinforce the embedded system controller nature of the new simulator, it is important to notice that the tool was designed with an architecture ready to enable the implementation of a distributed IcE (In Circuit Emulator), to remotely debug embedded systems running on physical devices over the internet, using the same user interface of the simulator. To implement this solution, the code responsible for executing the IOPT semantics is replaced with proxy code that forwards execution-flow commands to the remote embedded systems using HTTP requests. The status of the embedded devices is also read in real time using HTTP. The communication protocol presented in [6], and back-end HTTP server to run on the embedded devices are currently under development.

5 IOPT Petri Nets

The IOPT Petri net class is a non-autonomous Petri net class derived from Place-Transition nets [3], with the addition of input and output signals and events. In the same way as the traditional Petri nets, system behavior is expressed using places and transitions. However, the new input and output capabilities allows the definition of an interface with the external world, used to establish the communication between the controllers (IOPT models) and the controlled systems.

Figure 2 displays an example IOPT model. IOPT models support the usual places and transitions from P/T nets, plus the following characteristics:

1 – Input and output signals, can hold Boolean or integer range values, suitable to represent digital or analog I/O signals, respectively. 2 – Input events, representing instantaneous changes in input signals.

3 – Output events cause instantaneous changes in the value of output signals. Output events associated with transition firing, can increment or decrement the value of output signals by a specific amount. The signals hold the last values.

4 – Autonomous input and output events, not associated with I/O signals, are used to instantaneously propagate events between different component models.
5 – Guard expressions may be associated with transitions, containing logic conditions to inhibit transition firing according to the value of input signals
6 – Input events may also be associated with transitions to synchronize firing.
7 – Output actions may be associated with places, to define the value of output signals, when the places are marked. When no actions are active, the signals revert to a default value.
8 – Output actions associated to transitions, define the value of output signals when the transition fires. The affected signals memorize the last values.
9 – Transition priorities solve conflict situations where multiple transitions are simultaneously enabled, but there are not enough tokens to fire all of them.
10 – Test Arcs prevent the firing of transitions in the same way as normal Arcs, but do not consume the tokens and thus, do not cause transition conflicts.
11 – Maximal step execution semantics to provide deterministic execution, ensuring that all transitions ready to fire, will fire in the next execution step. Single server semantics is used, which means that even if a transition is enabled for multiple firings, only one firing will be produced per execution step.

6 Simulator Functionality

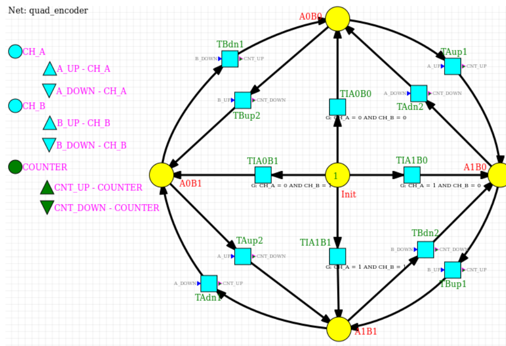

Fig. 2. IOPT Model - Quadrature decoder

Figure 3 presents the IOPT simulator user interface, composed by a toolbox on the left, a properties form on the right and a drawing of the model on the center. The toolbox contains icons for step-by-step execution, undo the last executed step, continuous execution, reset to initial marking, define a new forced marking, history navigation, history replay and history file export. The properties form contains the entire simulator status, including the current net marking, input signals, output signals and input and output events.

The model drawing reflects that status of the system, by changing the colors of the nodes. Places change colors according to the current marking: no tokens, one token or

many tokens. Transitions change color according to the readiness: autonomously enabled, ready to fire or both. Boolean input and output signals change color according to the value, and input and output events also flash when triggered. This color scheme helps the user interpret the current system status, resulting in much faster debug sessions. For example, instead of manually calculating the value of guard expressions to check if a transition is ready to fire in the next execution step, the user can simply observe the transition color.

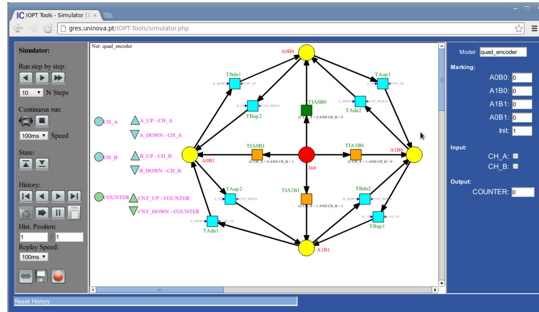

Fig. 3. Simulator / Debugger User Interface

The user can interact with the system both in the properties form and directly in the model drawing. For example, the value of an input signal can be changed on the corresponding item on the properties form, but it may also be changed by picking the corresponding signal on the graphic.

Continuous execution can be automatically interrupted with Breakpoints associated with transitions. Picking on a transition sets or resets a Breakpoint. A red border is drawn around transitions with Breakpoints. Execution stops immediately after a transition with a Breakpoint is fired, and the user interface reflects the system status after the firing. In case the user desires to inspect the status of the system prior to the firing, it is possible to undo the last executed step or move one position backwards in the recorded history.

The simulator maintains a complete record of the system status during entire simulation sessions. Whenever a new step is executed, the entire system status is recorded, including the value of input signals, events triggered, and information about transitions fired. The user can navigate through the saved history, and the user interface will change to reflect the status of the selected history step. This way, it is possible to inspect the past system evolution to detect flaws that may have not been noticed by the user during the original execution.

To simplify the inspection of the past system evolution, it is possible to replay the recorded history with programmable speed, starting on the current history step. In alternative, the simulator history can also be presented in the form of a spreadsheet table, as displayed in figure 4. To simplify inspection, repeated execution steps with identical status are compressed in a single table line with indication about the number of repetitions.

Finally, the history table can be exported in CSV (Comma Separated Values) text file, suitable for spreadsheet edition, or to be imported by other software. Taking advantage of the graphing functionality offered by most spreadsheet applications, it is very easy to create waveform diagrams of the input and output signals, as well as the internal system status variables.

For more detailed information about the simulator usage, tool options, interaction and color schemes, a user manual can be downloaded from the tools web interface [4].

7 Simulator Architecture

The core of the simulator is a based on a new automatic code generator tool that produces Javascript code. Instead of relying on a model interpreter, the part of the simulator code that implements the model semantics is dynamically generated each time the simulator is executed, with code that implements the semantic rules of execution defined in each model.

IOPT Simulator History

Model: quad_encoder

		Marking	Transitions	Input_signals		Output_signals	Input_events	Output_events
Step	Rep.		fired	CH_A	CH_B	COUNTER		
1	1	A0B0=1	TIA0B0	0	0	0		
2	1	Init=1	TIA0B0	0	0	0		
3	1	A0B0=1	TIA0B0	0	0	0		
4	5	A0B0=1		0	0	0		
9	1	A1B0=1	TAup1	1	0	1	A_UP	CNT_UP
10	2	A1B0=1		1	0	1		
12	1	A1B1=1	TBup1	1	1	2	B_UP	CNT_UP
13	1	A1B1=1		1	1	2		
14	1	A0B1=1	TAdn1	0	1	3	A_DOWN	CNT_UP
15	2	A0B1=1		0	1	3		
17	1	A1B1=1	TAup2	1	1	2	A_UP	CNT_DOWN
18	1	A1B1=1		1	1	2		
19	1	A1B0=1	TBdn2	1	0	1	B_DOWN	CNT_DOWN
20	2	A1B0=1		1	0	1		
22	1	A0B0=1	TAdn2	0	0	0	A_DOWN	CNT_DOWN
23	2	A0B0=1		0	0	0		
25	1	A0B1=1	TBup2	0	1	1023	B_UP	CNT_DOWN
26	2	A0B1=1		0	1	1023		
28	1	A1B1=1	TAup2	1	1	1022	A_UP	CNT_DOWN
29	2	A1B1=1		1	1	1022		
31	1	A1B0=1	TBdn2	1	0	1021	B_DOWN	CNT_DOWN
32	3	A1B0=1		1	0	1021		

Fig. 4. Recorded history table

The automatic code generator itself, is implemented using a XSLT Transformation [18] applied to the model's PNML [19] files. The code generator uses the same principles as the VHDL and C code generators and detailed information about the code generation techniques can be found in [9]. The generated Javascript code contains the following items:

- A semantics execution function that runs a single execution step of the model being simulated
- Several functions that evaluate the state of each transition, to check if it is autonomously enabled or the associated input events and guard functions are ready (to provide animated graphical feedback)
- Several JSON [Javascript Object Notation] objects representing the net marking, the input signals, output signals, input events, output events and model arrays.

The user interface code calls another XLST transformation to create an SVG (Scalable Vector Graphics) [20] document containing the graphical representation of the model, that is displayed in the central area of the simulator window.
To process each execution step, the simulator executes the following tasks:

a) Update the JSON objects representing input signals and autonomous input events
b) Update the user interface color schemes, using the transition evaluation functions
c) Call the Javascript function that executes a single execution step
d) Update the user interface status form, according to the new values on the JSON objects
e) Update the graphical model according to the new status and apply color schemes

This architecture is ready to implement an in-circuit emulator used to remotely debug embedded systems designed using IOPT tools. To achieve this, the Javascript execution function in c) is replaced by a proxy function that communicates with the remove system, sending and receiving the JSON objects used in a) and d).

8 Conclusions and Future Work

The simulator has been developed and integrated in the IOPT tools service and has been used to debug and simulate numerous models. The behavior of the simulated models has been compared with real world implementations of the same models running on embedded hardware boards, and also with the state-space graphs, with no inconsistencies detected. As a consequence, the simulator can be used effectively to detect design flaws before reaching the prototype implementation phase. As the typical development cycle takes only a few seconds after completing a model edition to start a new simulation session, development time can be enormously reduced. The alternative solution, debugging models using prototype boards, generally impose much longer development cycles, as the FPGA synthesis tools usually take many minutes to generate the bit-stream files necessary to reconfigure the hardware devices.

The rapid detection of design flaws often requires the simulation of the entire embedded systems. Although IOPT nets have been designed to the development of controllers, the user can build enhanced models containing both the controller and also the controlled systems. With this solution, the user can simulate the entire embedded systems, including controlled hardware and can even calculate the state-space graphs of the entire system.

Future work includes the addition of a waveform editor, to pre-program sequences of input signals and graphically visualize the resulting output signal waveforms. A database of signal waveforms can be stored in the cloud, and used as an automatic unit-test framework to perform regression tests and compare results with previous executions. An in-circuit emulator to debug remote systems is currently under development. Integration with an Animator tool is planned and the Simulator is currently ready to interact with other windows that present Animator generated screens. These screens will add new user interface windows, bringing additional user friendliness and the ability to be used by persons without knowledge about Petri nets.

Acknowledgments. This work was partially financed by Portuguese Agency" FCT - Fundação para a Ciência e a Tecnologia" in the framework of projects PEst-OE/EEI/UI0066/2011 and PTDC/EEI-AUT/2641/2012.

References

1. Pereira, F., Moutinho, F., Gomes, L.: IOPT-Tools - Towards cloud design automation of digital controllers with Petri nets. In: ICMC 2014 International Conference on Mechatronics and Control, July 3-5, Jinzhou, China (2014)
2. Gomes, L., Barros, J., Costa, A., Nunes, R.: The Input-Output Place-Transition Petri Net Class and Associated Tools. In: Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN 2007), Vienna, Austria, July 2007
3. Reisig, W.: Petri nets: an introduction. Springer, New York (1985)
4. Pereira, F., Moutinho, F., Gomes, L.: IOPT Tools User Manual - Version 1.1. FCT/UNL, Lisbon (2014). http://gres.uninova.pt
5. Zakas, N.C., McPeak, J., Fawcett, J.: Professional Ajax, 2nd edn. Wiley (2007) ISBN 1-4571-0715-5
6. Pereira, F., Gomes, L.: Minimalist Architecture to Generate Embedded System Web User Interfaces. In: Camarinha-Matos, L.M., Tomic, S., Graça, P. (eds.) DoCEIS 2013. IFIP AICT, vol. 394, pp. 239–249. Springer, Heidelberg (2013)
7. Pereira, F., et al.: Web based IOPT Petri net Editor with an extensible plugin architecture to support generic net operations. In: IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society. IEEE (2012)
8. Pereira, F., Moutinho, F., Gomes, L.: Model-checking framework for embedded systems controllers development using IOPT Petri nets. In: 2012 IEEE International Symposium on Industrial Electronics (ISIE), pp. 1399–1404. IEEE, May 2012
9. Pereira, F., Gomes, L.: Automatic synthesis of VHDL hardware components from IOPT Petri net models. In: IECON 2013-39th Annual Conference of the IEEE Industrial Electronics Society, pp. 2214–2219. IEEE, November 2013
10. Wiśniewski, R., Stefanowicz, Ł., Bukowiec, A., Lipiński, J.: Theoretical Aspects of Petri Nets Decomposition Based on Invariants and Hypergraphs. In: Park, J.J.H., Chen, S.-C., Gil, J.-M., Yen, N.Y. (eds.) Multimedia and Ubiquitous Engineering. LNEE, vol. 308, pp. 371–376. Springer, Heidelberg (2014)
11. Sadilek, D.A., Wachsmuth, G.: Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 63–78. Springer, Heidelberg (2008)
12. http://www.mathworks.com/products/simulink/
13. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Pratical Use, vol. 1. Basic Concepts. Springer, Berlin (1997)
14. Hamez, A., Hillah, L., Kordon, F., Linard, A., Paviot-Adet, E., Renault, X., Thierry-Mieg, Y.: New features in CPN-AMI 3: focusing on the analysis of complex distributed systems. In: Sixth International Conference on Application of Concurrency to System Design, ACSD 2006, June 28–30, pp. 273–275 (2006). doi: 10.1109/ACSD.2006.15
15. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L.: Renew – User Guide, University of Hamburg, Dept for Informatics, Theoretical Foundations Group, Rel. 2(2) (August 28, 2009)
16. Gomes, L., Lourenco, J.: Rapid Prototyping of Graphical User Interfaces for Petri-Net-Based Controllers. IEEE Transactions on Industrial Electronics 57, 1806–1813 (2010)

17. Programmable controllers - Part 3: Programming languages. IEC International Standard 61131-3 ed3.0, 2013-02-20
18. Tidwell, D.: XSLT - Mastering XML Transformations, 2nd edn.. O'Reilly Media (June 2008) ISBN 978-0-596-52721-1
19. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, Technology, and Tools. In: van der Aalst, W.M., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 483–505. Springer, Heidelberg (2003)
20. Scalable Vector Graphics (SVG) 1.1, 2nd edn., W3C Recommendation (August 16, 2011) http://www.w3.org/TR/2011/REC-SVG11-20110816/