From Petri net models to C implementation of digital controllers

Luis Gomes1,2, Rogério Rebelo1, João Paulo Barros2,3, Anikó Costa1,2, Rui Pais1,2,3
lug0@fct.unl.pt, rogeriorebelo@gmail.com, jpb@uninova.pt, ake@fct.unl.pt, ruipais@uninova.pt
1Universidade Nova de Lisboa 2UNINOVA 3Instituto Politécnico de Beja
Faculdade de Ciências e Tecnologia Centro de Tecnologia e Sistemas Escola de Sup. Tecnologia e Gestão
PORTUGAL PORTUGAL PORTUGAL

Abstract - This work presents a tool for automatic generation of controllers’ implementation code from Petri nets models amenable to be deployed into common platforms using widely used high level programming languages, such as C, C++, and Java. The generated code is linked with platform specific functions, supporting different types of implementation platforms, ranging from low-cost microcontrollers to workstations, and including microcontroller IPs (Intellectual Property) to be embedded into FPGAs (Field Programmable Gate Arrays). The system controller behavior is modeled using IOTP (Input-Output Place-Transition) Petri Nets models, which are represented through PNML (Petri nets Mark-up Language) notation. A tool for automatic code generation was developed, which achieved this goal in cooperation with other developed tools within a model-based development framework. Application to an automation system composed by a set of distributed controllers is presented.

I. INTRODUCTION

Petri nets are an abstract specification language with a well-known graphical representation, especially interesting for the construction of concurrent models. Yet, despite the availability of numerous tools [1], Petri nets remain relatively underused as a tool for model driven development probably due to the extremely scarce number of code generation tools. Hence, Petri nets are mostly used for the development of high-level models that are used to validate, and sometimes verify, systems. Latter, some of these maybe implemented as a separated task using distinct languages and tools.

A model-based development framework for the use of Petri nets as the design language for the construction of executable models, as presented in [2], will be used in this work as the reference. These can be validated using the token-player paradigm, but the tool here presented emphasizes the possibility of generating executable models for several distinct platforms, with different memory and processing capabilities. More specifically, we present a tool for the automatic generation of controllers’ implementation code from Petri nets models. The generated code is amenable to be deployed into common platforms using widely used high level programming languages, such as C, C++, and Java. The generated code is linked with platform specific functions, supporting different types of implementation platforms. These go from low-cost microcontrollers to workstations, and include microcontroller IPs (Intellectual Property) to be embedded into FPGAs (Field Programmable Gate Arrays).

II. DEVELOPMENT FLOW

In this section, the proposed embedded systems development flow adheres to the model-based development attitude, where Petri nets play the role of supporting formalism allowing interoperability between the different tools. The embedded systems development flow satisfies the following sequence of actions:

- Description of the controller system requirements through use cases, where the different functionalities of the system are identified;
- Modeling of the system’s behavior through IOTP Petri nets models [2], which are a class of Petri nets (PN) [3];
- Representation of the IOTP PN model using PNML notation;
- Automatic generation of code to implement the controller considering a specific implementation high-level programming language, which could be (in current stage of development) C, C++, or Java;
- Deployment of the previously generated code into the selected implementation platform after linkage of specific platform dependent code to assure data acquisition and proper output actuators activation.

The development is done using a variety of tools that interact through files using a neutral format, the PNML notation, as shown in Figure 1 (most of these tools were developed within the project FORDESIGN - http://www.uninova.pt/fordesign).

The graphical editor and simulator (tool Snoopy-IOTP [4]) allows the editing of the IOTP PN models and generates their representations into PNML format [5]. Two tools are in place
to allow model composition: (1) the \textit{OPNML2PNML} tool [6], allowing reusability of models already available and the \textit{Split} tool [7] for model decomposition allowing the splitting of the model into several concurrent models targeting distributed execution of IOPT models.

Several automatic code generators are available for specific languages. The automatic code generator \textit{PNML2VDHL} [8] automatically generates the VHDL code of the controller system from the PNML file, while the tool \textit{PNML2C} supports the generation of C code.

Finally, the tool \textit{Configurator} allows the generation of code to be deployed into a specific implementation platform, including microcontrollers from PIC family, MicroBlaze microcontroller IP and hardware descriptions for Xilinx FPGAs Spartan-3 and Virtex-II. The tools supporting the automatic generation of C++ and Java code are not yet fully integrated in the \textit{Configurator} tool flow.

It is important to note that the \textit{Snoopy-IOPT} tool allows hierarchical organization of the IOPT PN models. However, the tools for code generation operate from the flat IOPT PN model (which can also be generated by the graphical editor).

![Figure 1 - Tools overview for Petri nets-based model-based development flow.](image-url)

\section*{III. The Input-Output Place-Transition Petri Nets Class}

In this section, we briefly present the main characteristics of the Input-Output Place-Transition Petri net class (IOPT). A complete presentation, including the formal semantics and the Petri net type definition [5], was already presented elsewhere [2].

The IOPT class is an extension of place-transition nets [3]. The objective is to allow the specification of the interaction between the environment and the net, which models a controller.

To that end, the controller model specifies input and output signals and events. As these restrict the model behavior, the net becomes non-autonomous. Several other non-autonomous net classes exist (e.g. [9] [10] [11] [12] [13]) some for quite some time, but none as resulted in a framework supporting code generation for specific platforms using general purpose high-level programming languages.

The synchronized behavior implies that net firing is only possible at specific instants named tics, defined by an external global clock. The IOPT nets have maximal step semantics: each step includes all the autonomously enabled transitions whose guards and input events are true. The step is executed between two tics.

Compared to place-transition nets, IOPT nets have the following additional possibilities, which are briefly presented after:

- Test arcs and test arcs weights;
- Priorities in transitions;
- Guards in transitions; the guards are Boolean functions of the input signals;
- Input and output events in transitions;
- Output signals in places.

Test arcs have the usual firing semantics: if a transition is connected to a place through a test arc than the transition can only fire if the place has, at least, as many tokens as the test arc weight. Yet, when transition is fired the test arc does not remove any tokens. So, when regular arcs and test arcs are output arcs of a specific place, no conflict is produced from test arcs.

The priorities in transitions allow determinism when firing transitions in conflict. Hence, typically, all transitions in structural conflict are given a priority value. Among those enabled transitions, only the transition with the maximum priority is fired (as long as stays enabled after the firing of other transitions in the same conflict set with higher priorities).

The transition guards are Boolean functions of the external input signals. A transition can only be enabled if its guard evaluates to true.

Each transition can have an input and an output event. Both are optional. If present, the input event must be true for the transition to fire. The output event is made true when the transition fires.

Each place can also have a set of rules. Each rule has three parts: (1) a Boolean function of the net marking, (2) an output signal, and (3) a value. After each step, for each rule, if the condition is true the respective signal is assigned the value.

Next, we present how to generate executable code from a IOPT model.

\section*{IV. Code Generation Strategy}

The generated code has two main code components: (1) the platform independent executable model (the controller); (2) the platform dependent code. The first is a direct translation of the net model, while the second handles the communication between the controller and the environment. Figure 2 outlines this architecture. The executable model
interacts with the platform dependent code, namely the input and output interfaces.

The communication between the execution model and the platform dependent code is based on Signals and Events. The Input Interface reads input signals and events from the respective external physical inputs. The input signals are read and stored in a data structure shared with the executable model code. The Input Interface code defines new events by comparing the present signals values with the ones from the previous step.

In a similar way, the output interface code interacts on the physical outputs of the specific platform. Currently, the platform dependent code is manually written, but the objective is to automatically generate it based on the hardware description of the needed functionalities.

The execution algorithm has three main parts:
1. Input reading
2. Internal Processing
3. Output updating

Parts 1 and 3 are executed by the platform dependent code, while part 2 corresponds to the net execution.

The behavior associated to each of these three parts was already presented elsewhere [14, 15]. Two auxiliary structures allowing coherent execution of Petri net semantics were used: Auxiliary Marking used as an initial copy of Current Marking, from where tokens are removed after one transition firing; and Generated Marking that accumulates tokens generated during one execution step; at the end of the execution step, the Current Marking is updated with sum of Auxiliary Marking and Generated Marking. In summary, the step execution to be carried by the implementation platform will consider the following steps (in reference to Figure 2):

1. reset ISDS and IEDS
2. initialize Current Marking
3. begin execution step
4. input acquisition by Input Interface
5. reset ODS and OEDS
6. initialize Auxiliary Marking with Current Marking
7. reset Generated Marking
8. net firing and OEDS writing
9. update Current Marking
10. update ODS based on Current Marking
11. output writing by Output Interface
12. end execution step

In the following, a brief description of some of the algorithms used by the code generator tool to generate the C code is presented.

The code generator tool receives a single PNML file and creates five files: main.c, functions.h, functions.c, netc.h, and netc.c.

The main file provides an interface to the user. The functions files include functions that generate input events from input signals, functions that generate output signals from output events, and functions that generate output signals based on the current net marking.

The netc files contain the net execution code. This includes two main functions: (1) start, which initializes all variables with the Petri net model initial values; (2) run, which executes a net step.

Coming to the execution of a net step, it is important to stress that:

- Each place is represented by three variables:
  - The first variable (Current Marking in Figure 2) is used to store the number of tokens in the beginning of iteration.
  - The second variable (Auxiliary Marking in Figure 2) is initialized with number of tokens in the beginning of iteration and it is from where tokens are removed when they are consumed due to the firing of a transition.
  - The third variable (Generated Marking in Figure 2) is initialized with zero and it is where the created tokens are added.

- At the end of the iteration, the Generated Marking variable is added to the Auxiliary Marking and saved into Current Marking.

- The adopted net execution strategy relies on analysis and firing of transitions ordered by their associated priorities (which were specified by the modeler).

It is important to note that different ways to transpose the transitions list correspond to different conflict resolution strategies. The presented approach using priorities in transitions imposes an a-priori decision to solve conflicts. Alternatively, a pseudo-random ordering would mean a fair resolution strategy. As it was already stressed in [14], the referred strategy for conflict resolution based on the ordering on priorities is very effective for software implementations.

As referred, the adopted net execution strategy for step execution starts by looking to a list of ordered transition. Transitions with higher priority are handled first (and so the conflicts are intrinsically solved). Setting up the list of transitions to be analyzed is accomplished using the following algorithm (here referred only for completeness purposes):

13. Reset Actual Priority
14. Reset Number of Transitions
15. while Number of Transitions < Total Transitions do
16. Increment Actual Priority
17. For all transitions do

Figure 2 – Execution Framework, where:
- I.S.D.S – Input Signal Data Structure
- I.E.D.S – Input Event Data Structure
- O.S.D.S – Output Signal Data Structure
- O.E.D.S – Output Event Data Structure
- Gen. Mark. – Generated Marking
- Aux. Mark. – Auxiliary Marking
- Curr. Mark. – Current Marking
18. If Transition Priority equals Actual priority then
19. (Add Transition to the execution list)
20. Endif
21. Endfor
22. Endwhile

For adding a new transition to the execution list, the following elements are created: a list of source places; a list of source arcs; a list of target places; and a list of target arcs. Conditions to be evaluated for firing the transition include marking enabled, signal input guard and event evaluations.

The following algorithm partially illustrates part of the process for code generation at this stage:
1. Reset Source Places List
2. Reset Source Arcs List
3. Reset Target Places List
4. Reset Target Arcs List
5. Add Text "if("  
7. If is arc test then
8. Add Text "p_source place id_Mark == Source Arc inscription"
9. Else
10. Add Text "aux_p_source place id_Mark == Source Arc inscription"
11. Endif
12. Endfor
13. For all Signal Input Guards do
14. If Language is C then
15. Add Text "&& Signal input gard text"
16. Endif
17. Endfor
18. For all Input Events do
19. Add Test "&& InputEvent == 1"
20. Endfor
21. Add Text ")"
22. (Add associated firing actions)

The associated firing actions can be described by the following algorithm, where tokens are removed and output events are generated according to the firing semantics.
1. For all Source Places do
2. If isn’t arc test
3. Add Text "aux_p_source place id_Mark += Source Arc inscription"
4. Endif
5. Endfor
6. For all Target Places do
7. Add Text "aux_p_target place id_Mark -= Target Arc inscription"
8. Endfor
9. For all Transition Output Events do
10. For all Output Events do
11. If Transition Output Event is Output Event Then
12. If edge is up Then
13. Add Text "aux_Signal += Event Level"
14. Else if edge is down Then
15. Add Text "aux_Signal -= Event Level"
16. Endif
17. Endif
18. Endfor
19. Endfor
20. Endfor

At the end of each step, the net output signals are generated based on the current net marking.

V. APPLICATION EXAMPLE

To illustrate the applicability of the above presented framework, an application example introduced by [10] will be used. The goal is to produce the controller for an automation system composed by three cars to carry goods (see Figure 3). The cars (wagons) are moving forward and backward between two end points (A and B). Each wagon has its own trajectory and velocity, but they should start moving at the same moment. They start moving forward when all of them are at home position, points A[i], and the bottom GO is pressed. The backward movement should start when all wagons are on its end position, points B[i], and the bottom BACK is activated. In this sense, the system controller has six presence detector sensors as input signals (three at the initial position A[i] and three at the end position B[i]), as well as bottoms GO (to initiate the forward moving) and BACK (in order to initiate the backward movement). As outputs, the controller will have two sets of signals: one of them to control the engines of the three cars (M[i]), and the other one indicating the direction of the movement (Dir[i]), as shown in Figure 3.

In order to use Petri nets as a system specification language, the IOPT model presented at Figure 4 was produced. This model can be used to automatically generate implementation code for the controller of the system. However, if the objective is to obtain a distributed controller, it is necessary to split the IOPT model into three sub-models, each one of them to be deployed into a local controller installed in each car. To achieve this objective we can use the net splitting operation and associated Split tool [7], as already referred. Figure 4 identifies the transition nodes GO and BACK to be used as cutting set when applying the net splitting operation.

![Figure 3 - Application example: automation system for the control of three wagons.](image)

![Figure 4 – IOPT Petri net model of the application example.](image)
After applying the net splitting operation, three sub-models are generated, as presented in Figure 5. Each sub-model has some dedicated input and output signals (associated with its own environment), as well as a set of input and output events to assure communication with other sub-models. This communication among the sub-models is assured through directed synchronous communication channels. Those communication channels assure the synchronous firing of a set of transitions; one of the transitions in the set involved in each communication channel receives the attribute of master and is responsible for generating an output event; another transition involved in the same communication channel will receive the attribute slave and is receptive to the input event generated by the master transition. In Figure 5 each communication channel has a unique associated number.

The C code associated with the presented models, both for centralized execution (Figure 4) as well as for distributed execution (Figure 5), was successfully generated using the automatic code generator.

There are three types of controllers: one associated with the centralized controller (only one controller for the three cars), another one associated with the “master” car, and the third one associated with the “slave” cars in the distributed controller.

The number of lines of automatic generated code for centralized controller is 177 lines (34 lines of initialization and 143 of execution). On the other hand, the number of lines of automatic generated code for distributed controllers is 75 lines (16 lines of initialization and 59 of execution) to “slave” cars (car 2 and car 3 in Figure 5) and 121 lines (24 lines of initialization and 97 of execution) to “master” car (car 1 in Figure 5), respectively.

The referred generated code was successfully deployed into microcontrollers from PIC family (PIC18F4620), after linkage with the platform dependent code, allowing assigning inputs and output signals to physical pins of the devices.

VI. BRIEF DISCUSSION

Writing code by hand can improve and optimize a specific system, producing better performing machine code than the one produced by automatic tools. However it may have many disadvantages: it requires more developers with higher expertise and a larger development time; it is more susceptible to errors and it is difficult to maintain. We can compare it to a low level language: it performs better but it is harder to develop.

On the other hand, automatic code generation has important advantages the generated code always increases in quality over time, it is consistent in structure and styling rules, and it can be produced in a fraction of time.

Comparing the performance of hand written code and automatically generated code is a double-edged sword. Code produced by hand can excel on a specified platform, but it is not easily adapted to a new platform.

The optimization factor is especially relevant in embedded systems, were memory resources and execution speed are limited, requiring a delicate balance between their usage.
On the developed code generation tools, optimization factors and the compatibility with different architectural platforms were seen as fundamental goals. Several optimizations techniques were used exploring different factors associated with the IOPT Petri net class and its properties, compiler options, and implementation platform specificities. As one example, one can refer to dichotomy between usage of data memory and code memory considering the specificities of the implementation platform.

It is important to note that one important characteristic of code generation tools usage is that it is easy to test different optimization factors without too much effort, time, and cost.

VII. FUTURE WORK

With the increasing number and complexity of development systems for embedded platform, modeling languages and tools become an important factor in the Systems Development Life Cycle (SDLC). The Systems Modelling Language (SysML) is an optimized notation for Systems Engineers. It was created to improve communications across the SDLC, increase the re-use of designs, and lower maintenance costs [16]. The use of tools could enable model simulation and execution, permitting early verification of designs, thus increasing software quality and reducing system development risks.

Future efforts will be driven to improve the presented system with a conversion tool from SysML model to IOPT PN model. With this improvement, it will be possible to generate code from SDLC, thus allowing model verification, simulation, and execution. That development effort will be supported by the Eclipse Modeling Framework (EMF), which implements most of the Model-Driven Architecture (MDA) core specifications.

EMF can be used to support a wide variety of applications, including SDLC support, model transformations and systems integration. This platform is a strong candidate to integrate many of the developed tools as well new functionalities on the model transformations.

VIII. CONCLUSIONS

The presented development flow is based on model-driven development where IOPT Petri nets have the main role. Several tools were developed. These include an IOPT editor, tools allowing model compositions and decompositions and several automatic code generators. The common base for those tools is the PNML representation for the IOPT Petri net models. This is one of the most important advantages of our embedded system development method: the PNML representation provides the basis for the automatic code generators and their interoperability with a set of other tools devoted to analysis, design, verification, simulation, and deployment into specific platforms.

REFERENCES


