A Tool for Tailored Code Generation from Petri Net Models

Rui Pais\textsuperscript{1,2}, João Paulo Barros\textsuperscript{1,2}, and Luís Gomes\textsuperscript{1}
\textsuperscript{1}Universidade Nova de Lisboa / UNINOVA, Portugal
\textsuperscript{2}Instituto Politécnico de Beja, ESTIG, Portugal
\{ruipais, jpb, lugo\}@uninova.pt

Abstract

The use of Petri nets for the modelling of discrete-event systems is well-studied. Yet, the tools allowing the implementation of these models, and supporting code generation, are still very few, almost non-existent. This paper starts by presenting a Petri net class, based on place/transition nets and well-known concepts from synchronised and interpreted Petri nets. This Petri net class allows the association of external input signals to transitions and the association of external output signals to transitions and place markings. Additionally, the class provides support for the specification of input and output events. Next, the paper presents a code generator able to output optimised executable code from these nets. The generated code can be optimised by several distinct strategies, which facilitate the creation of code tailored to specific platforms, as well as for specific classes of Petri nets.

1. Introduction

A common way to model embedded systems uses a decomposition based on a controller that interacts with its environment. By using a synchronising signal (a global external clock), this interaction can be forced to happen at periodic points in time. These discrete event systems can be conveniently modelled by Petri nets and this fact has been known for quite a long time (see, for example, the article by Holloway et al. [6]). Notwithstanding this known adequacy, the number of code generators from Petri net models is extremely scarce as attested by the Petri Nets Tool Database [7].

This paper presents a code generator from Petri net models that is able to generate highly configurable code, amenable to be executed even in platforms with minimal resources. To that end, it relies on a database of possible code blocks, each one implementing a slightly different code-level design option. These blocks are chosen and configurable based, not only on the specific properties of the Petri net class in use, but also on the particular characteristics of each model. Finally, the user has control over another set of code generation options. When taken together, all these options allow the creation of code highly adapted to the platform where it will be executed.

Presently, the code generator is able to handle a simple and intuitive extension to place/transition nets [9], arguably the most used and well-known Petri nets’ class, allowing its use for the specification of discrete event systems’ control. This class is named input/output place/transition net (IOPT). It adds a minimal set of notations to place/transition nets allowing their connection to the environment. More specifically, IOPT nets allow the specification of external input signals and external input events associated to transition firing. They also support the specification of external output events and external output signals (actions). The former are associated to transitions, while the latter are associated to place markings.

The paper is structured as follows: Section 2 briefly presents a framework for the use of the code generator tool; Section 3 defines the IOPT Petri nets class; Section 4 presents the used code generation strategies, and Section 5 focus on the supported optimisations. Section 6 presents an example model and shows some samples of the generated code. Finally, Section 7 concludes and specifies points for future work.

2. A development environment

The code generation from IOPT models is part of a development environment, which besides the code generator application here described and named PnGenerator, will include a graphical editor, a simulator and a verification tool. The development of the graphical editor is foreseen but not yet started. The latter two tools are closely connected to the PnGenerator and are currently being developed. The complete system is presented in Fig. 1, which can also be found in two preliminary papers where the complete vision for the development environment was presented [1, 4]. The editor (the PnEditor in the top-left corner of Fig. 1) will be able to generate Petri net models in the PNML language, an XML-based interchange format for all Petri net classes [2, 8]. Presently, the PnGenerator reads this format and outputs ANSI C code. Future developments will include the generation of code using different languages, like VHDL code for hardware.
implementation, SystemC for system-level designs, and languages for Programmable Logic Controllers programming. Finally, the code created by the PnGenerator tool is currently used for the final execution in a specific hardware platform, but its use is foreseen also for simulation and analysis purposes: through the addition of appropriate interface code, the generated executable code will also be used to simulate and verify the model.

3. A Petri net class integrating control specification

As already stated, the modelling of discrete-event systems is a typical application for Petri nets [6]. One of the most used strategies is the logic controller approach where the net models the controller. Hence, the net has to communicate with the surrounding environment. More specifically, the Petri net is extended with new constructs to allow the modelling of external signals, events, or both. This is the approach by David and Alla and also by Silva in their proposals for synchronised and interpreted Petri nets [3, 10]. Here, we present a class of non-autonomous Petri nets, named input/output place/transitions nets (IOPT).

The class of IOPT nets is an extension to the well-known class of place/transition nets [9]. The extensions are all related to the non-autonomous part: controller inputs can be associated to transitions; and controller output can be associated to places, transitions, or both. The design of this new net class, which uses known concepts from literature, had two main objectives: (1) to maximise the syntactic and semantic similarity to a well-known autonomous Petri net class, thus minimising the addition of new constructs and concepts; (2) to allow a simple and effective connection to the controller’s environment.

3.1. A model for the controlled system

From the controller point of view, we consider that all systems, which are going to be controlled can make available a set of readable input signals, and a set of writable output signals. Additionally, we define input events that model modifications between consecutive input states. We also define output events, which are activated by transition firing.

Definition 1 (System interface) The interface of controlled system with a IOPT net is a tuple IS = (IS, IE, OS, OE) satisfying the following requirements:

1. IS is a finite set of input signals.
2. IE is a finite set of input events.
3. OS is a finite set of output signals.
4. OE is a finite set of output events.
5. IS ∩ IE ∩ OS ∩ OE = ∅.

The input events are computed based on changes of the associated input signals. Yet, this computation is made transparent to the net model, which sees the input events as just another kind of input signal. This is made possible by external code that establishes an interface between the data structures used by the net and the external input signals. The same approach is followed with respect to output signals. This interface is illustrated at the bottom of Fig. 1 (Input Interface and Output Interface) and was explained in detail in a previous paper [4].

Definition 2 (System input state) Given an interface ICS = (IS, IE, OS, OE) with a controlled system (Def. 1), a system input state is defined by a pair SIS = (ISB, IEB) satisfying the following requirements:

1. ISB is a finite set of input signal bindings: ISB ⊆ IS × N₀.
2. IEB is a finite set of input event bindings: IEB ⊆ IE × B.

Next we define the class of IOPT nets.

3.2. Input/output place/transitions nets

The IOPT nets definition assumes the availability of an inscription language allowing the specification of algebraic expressions, variables, and functions. The set of boolean expressions is named BE and the function Var(E) returns the set of variables in a given expression E. As the implementation is the final objective of a IOPT model the inscription language should preferably be the same as the generated code language (e.g. ANSI C), as this simplifies the code generation process and avoids the introduction of an additional language in the development process.

Definition 3 (IOPT net) Given a system to be controlled with an interface ICS = (IS, IE, OS, OE), a IOPT net is a tuple N = (P, T, A, M, weight, isg, ie, oe, osc) satisfying the following requirements:

1. P is a finite set of places.
2. T is a finite set of transitions (disjoint from P).
3. A is a set of arcs, such that A ⊆ ((P × T) ∪ (T × P)).
4. M is the marking function: M : P → N₀.
5. weight : A → N₀.
6. isg is an input signal guard partial function applying transitions to boolean expressions (where all variables are input signals): isg : T → BE, where ∀eb ∈ isg(T), Var(eb) ⊆ IS.
7. ie is an input event partial function applying transitions to input events: ee : T → IE.
8. $oe$ is an output event partial function applying transitions to output events: $oe : T \rightarrow OE$.

9. $osc$ is an output signal condition function from places into sets of rules: $osc : P \rightarrow \mathcal{P}(RULES)$, where $RULES \subseteq (BES \times OS \times \mathbb{N}_0)$, $BES \subseteq BE$ and $\forall e \in BES, Var(e) \subseteq ML$ with $ML$ the set of identifiers for each place marking after a given execution step.

The external input signals are associated to transitions in two different ways: (1) directly as variables of a boolean expression that will act as a guard for transition firing; (2) indirectly as events that are computed based on signal modification.

External output signals can be changed in two different ways: (1) based on the place markings at the end of a step (Moore-like fashion); (2) based on transition firing (Mealy-like fashion).

Regarding the transition fire rules, the synchronised paradigm [3] is adopted: 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. From the net model point of view this means to choose a maximal step of execution (with a maximal number of transitions). The synchronised paradigm also implies that the net evolution is only possible at specific instants in time named $tics$. These are defined by an external global clock. An execution step is the period between two $tics$.

In the following definitions we use $M(p)$ to denote the marking of place $p$ in a net with marking $M$ and $\bullet t$ to denote the input places of a given transition $t$ or a given set of transitions $S$: $\bullet t = \{p|(p,t) \in A\}$; $\bullet S = \{p|(p,t) \in A \land t \in S\}$.

**Definition 4 (Enable condition)** Given a net $N = (P, T, A, M, weight, isg, ie, oe, osc)$ and a system interface $ICS = (IS, IE, OS, OE)$ between $N$ and a system with input state $SIS = (ISB, IEB)$, a transition $t$ is enabled to fire iff the following conditions are satisfied:

1. $\forall p \in \bullet t, M(p) \geq weight(p,t)$.

2. The transition $t$ input signal guard evaluates to true for the given input signal binding: $isg(t) < ISB$.

3. $(ie(t), true) \in IEB$.

**Definition 5 (IOPT net step)** Let $N = (P, T, A, M, weight, isg, ie, oe, osc)$ be a net and $ICS = (IS, IE, OS, OE)$ a system interface between $N$ and a system with input state $SIS = (ISB, IEB)$. Let also $ET \subseteq T$ be the set of all enabled transitions as defined by Def. 4. Then, $Y$ is a step in $N$ iff the following condition is satisfied:

$$Y \subseteq ET \land \exists t_1 \in (ET \backslash Y), \exists SY \subseteq Y, (\bullet t_1 \land \bullet SY) \neq \emptyset \land \exists p \in (\bullet t_1 \cap \bullet SY), (\text{weight}(p, t_1) + \sum_{t \in SY} \text{weight}(p, t) > M(p))$$
A IOPT net step is maximum. This means that no additional transition can be fired without becoming in effective conflict with some transition in the chosen maximal step. Finally, we define a IOPT net step occurrence and the respective successor marking.

**Definition 6 (Step occurrence and successor marking)**

Given a net $N = (P,T,A,M,\text{weight, isg, ie, oe, osc})$ and a system interface $ICS = (IS,IE,OS,OE)$ between $N$ and a system with input state $SIS = (ISB,IEB)$, the occurrence of a step $Y$ in net $N$ returns the net $N' = (P,T,A,M',\text{weight, isg, ie, oe, osc})$, equal to the net $N$ except for the successor marking $M'$ which is given by the following expression:

$$M' = \left\{ \left( p, m - \sum_{t \in Y \cap (p,t) \in A} \text{weight}(p,t) + \sum_{t \in Y \cap (t,p) \in A} \text{weight}(t,p) \right) \in (P \times \mathbb{N}_0) \right\}, (p,m) \in M$$

### 4. Code generation

It has to be stressed that there are important benefits on code generation:

**Quality** Manually written code varies in quality through the lifecycle of a project while, by improving the design, generated code always increases in quality over time; once bugs are found they can be corrected in the code templates.

**Consistency** Automatically generated code is consistent in structure, methods signatures, variable naming and comments.

**Productivity** Generators can build code in a fraction of the time compared to manually written code; this code can be used for test and execution.

#### 4.1. Code Generation Model

The adopted model for code generation, also named "Code Munger" [5], is the most common form of code generator. A code munger processes one or more source code files and generates one or more output files. In our application it reads one Petri Net file, in the PNML format [2, 8], and generates a set of ANSI C files based on predefined template code blocks.

One of the project’s key features is the compatibility with different architectural platforms. This is the reason for the use of ANSI C as the generated language, as it is usually available in any software-based platform.

To construct the generated source code, the PnGenerator (presented in the centre of Fig. 1) uses specific information about the Petri Net class in use and chooses from a large set of predefined code blocks for building the necessary functions and data type definitions. Also, several optimisation options allow the user to parameterise the generated code. Figure 2 illustrates the main inputs and output of the PnGenerator application, together with a high-level view of its internal architecture.

#### 4.2. Block Generator

The block generator (see Fig. 2) allows the modification of predefined code blocks outside of the application. It also simplifies the generation of similar code blocks. An XML tag `<id>` identifies each block. The generator receives an order to produce a specific block of code together with the necessary information. It processes the template block against received information and returns its result. Next, we illustrate the ID5450v1 and ID5450v2 template blocks. These provide two different implementations for the block with id=ID5450. These two blocks are equivalent as specified on the templates by the XML tag `<equivalent>`.

```c
//<id>ID5450v1</id>
//<id>ID5450v2</id>
const UCHAR * IO_inputEventBufferIdF[] = {

#define IO_inputEventId(trIndex) IO_inputEventBufferIdF[trIndex]  

switch(trIndex)  
    case #index#: { return "#(index)#"; }

```
This block requires the presence of the IO_IEB_INDEX_TYPE data type as specified through the XML.<require> ID5430v1</require>.

The Block Generator receives a parameters’ list containing the block name and the respective data needed to tailor the block generation.

As an example, the code of the two presented blocks is created by the following two methods (written in the C# programming language):

```csharp
protected virtual string generateID5450v1()
{
    const string blockName = "ID5450v1";
    string[] arraylist = {string[]}
        IO_inputEventBufferIdF.ToArray();
    return generateBlock(blockName, arraylist);
}

protected virtual string generateID5450v2()
{
    const string blockName = "ID5450v2";
    string[] arraylist = {string[]}
        IO_inputEventBufferIdF.ToArray();
    return generateBlock(blockName, arraylist);
}
```

As illustrated by the previous methods, the Block Generator uses the same array of "Input Event IDs" IO_inputEventBufferIdF to process the respective templates and generate the following blocks:

// Block: ID5450v1
const UCHAR * IO_inputEventBufferIdF[] = {
    "leaving",
    ...
    "permissionToExit"
};
#define IO_inputEventId(trIndex)
    IO_inputEventBufferIdF[trIndex]

// Block: ID5450v2
const UCHAR * IO_inputEventBufferIdF(IO_IEB_INDEX_TYPE trIndex) {
    switch(trIndex) {
        case 0: { return "leaving"; } 
        ...
        case 5: { return "permissionToExit"; } 
    }
    return NULL;
}
```

Templates can have some replaceable keywords to tailor code generation. All these keywords begin and end with the character #. The keywords used by the PnGenerator are explained next.

- `#0#` - this keyword is replaced by the first value sent to the generator. It accepts any number and replaces the corresponding keyword with the respective parameter. If the generator receives a parameter that is an array, the following keywords are used:
  - `#index#` - array index. The line that contains this keyword is repeated for all array indices.
  - `#[index]#` - array value on each index. The line that contains this keyword is repeated for all array values.
  - `[#FIRST]#` - first array value. When these keyword is present, the keyword `#[index]#` applies to all arrays indices except the first.
  - `[#LAST]#` - last array value. When these keyword is present, the keyword `#[index]#` applies to all arrays indices except the last.

Currently, the Block Generator uses a database of 110 Kbytes which contains approximately 550 code blocks.

5. Code optimisation

The well-known trade-off between speed and memory resources says that it is often possible to optimise memory usage at the expense of some extra CPU cycles, and vice versa. This is especially relevant in embedded systems as they usually have ad-hoc, and thus frequently minimal, memory resources. The same is true for execution speed. This fact may imply a delicate balance between the two criteria.

The generated data structures can be optimised based on the used Petri Net class. A modification of the data structure generally implies changes on the associated methods. Whenever possible, the access to the data structures is defined by an interface definition and an effort was made to have a common interface without a performance penalty. For example, the following code snippets use the same interface (TR_transitionId(trIndex)) for distinct data formats.

```csharp
const UCHAR * TR_IdF[] = { "Open",
    "Exit", ...
};
#define TR_transitionId(trIndex) 
    TR_IdF[trIndex]
```

```csharp
const UCHAR * TR_transitionId (TR_INDEX_TYPE trIndex) {
    switch(trIndex) {
        case 0: { return "Open"; } 
        ...
        case 5: { return "permissionToExit"; } 
    }
    return NULL;
}
```
Optimisation usually means the use of some techniques and tricks, which make code more difficult to read. Although the main goal is to generate optimised code, it is also possible to obtain highly readable code. This is important to understand the generated code, which can be very useful if the need to change it arises.

Next, we present a brief overview/discussion on some of the used code optimisations techniques. These are based on: (1) the Petri net class; (2) the Petri net class properties; (3) code based options; (4) compiler options; and (5) platform specificities.

5.1. Petri net class based optimisation

Since code is generated for a specific Petri Net Class, only the necessary data structures need to be generated. Especially due to optimisations related to execution speed, the generated code is tailored to the generated data structures. For example, as illustrated in Fig. 3 the needed data structures are different if arcs can have weights or not.

![Figure 3. Alternative data structures depending on arcs' weights.](image)

5.2. Petri net class properties-based optimisation

The properties of each Petri Net Class can provide useful information for the code generator. For example, when appropriate, bit based data types can be very useful for reducing memory usage.

As explained next, it is possible to optimise different data structures based on the known Petri net class properties.

Optimisations based on place properties If all places have a known limited capacity, the generator will produce information to represent place markings using the smallest possible numeric data type. If the place capacities vary widely, the places can be ordered by the marking data type capacity, thus improving memory usage.

Optimisations based on arc properties The simplification illustrated in Fig. 3 can also be done when all the arcs have a constant weight and no other arc information is used (name, identification, etc.). Arc weights storage can also be optimised just like suggested for place markings.

Optimisations of input or output events and signals

The generation of code to manipulate input or output events and input or output signals depends on their presence or not, in each concrete model.

Bit packing Petri Net models support various variable types, but, in many cases, they can be represented with smaller data types, thus requiring less data memory. Bit packing variables into a single integer or integer array allows the manipulation of larger clusters. Yet, this brings additional efficiency costs, namely the assigning of specific bits to or from the integers.

5.3. Code based optimisation

Compiler optimisations based on generated code are difficult to be made by hand and in many cases have an important impact on the overall system. Hand made optimisation are important when it is possible to get better solutions to solve the same problem.

Manual code optimisation Some manual code optimisations were made based on the generated code and many new solutions were tried for each piece of generated code. The following example illustrates one kind of optimisation sometimes used to minimise the size of the generated data structures. In this example, we consider one Petri Net with five arcs of weight 1, three arcs with weight 2 and one last arc with weight 5. The first implementation had one array with all the arc weights:

```c
const UINT8 IA_weightF[] =
{ 1, 1, 1, 1, 1, 2, 2, 2, 5};
#define IA_weight(inArcIndex)
IA_weightF[inArcIndex]
```

After hand made optimisation the previous arc information was reduced to the following definition, which avoids the array definition, thus reducing memory consumption:

```c
#define IA_weight(inArcIndex)
(inArcIndex < 5 ? 1 :
 (inArcIndex < 8 ? 2 : 5))
```

This solution is possible because all arcs were ordered in ascendant order of arc weights. The use of `#define` declarations allow uniform method interfaces and generate inline code thus providing efficient and readable code.

5.4. Compiler optimisation

Compiler optimisations can have an important impact on the overall system; nevertheless problem-specific optimisations can offer huge advantages. Hence, a combination of both is usually the best solution. To that end, it
is necessary to have a complete understanding of the used platform and the respective compiler. Next, we present a list of some optimisation techniques available in most compilers:

- Propagation (constants, types, copies), arithmetic simplification, constant folding, redundant check elimination (nulls/casts/array), simple inlining, pre-computation of constants, loop invariant code motion, loops fusion, etc.

- Common sub-expression elimination, expression simplification, redundant load elimination, aggressive inlining, global copy/constant propagation, scalar replacement, code reordering, etc.

- SSA (static single assignment form) based optimisations on scalars and arrays, redundant load and array bounds check elimination, loop optimisations, etc.

5.5. Platform specific based optimisation

Distinct hardware platforms have different configurations in terms of memory, processor speed, input/output ports, etc. As all these resources are typically scarce there are no obvious solutions for each problem. For a small Petri Net model, the generated code can be optimised to speed up using all memory resources. With big models memory optimisations become important.

6. Example

This section briefly presents a model and the respective generated code. The model is shown in Fig. 4 and specifies an access controller to a parking lot with one entrance and one exit. Regarding the used notation, the input and output events are represented by empty triangles pointing to and from the transitions, respectively. The input signal guards are presented between square brackets and the output event identifiers are underlined. In this example, all input events have the same identifiers as the respective transitions. The output signal conditions associated to places are presented inside boxes.

For the implementation platform we used a low-cost microcontroller, from the PIC family, with enough input/output pins for the task. This was complemented by a C compiler integrated in the microcontroller tools. For this model the generated code has around 900 lines of ANSI C code (1250 lines including comments) and uses about 180 generated blocks.

The following code samples (slightly edited for presentation purposes) illustrate the structure of the generated code:

```c
// Block: ID1020v1
#define PL_MARKING_TYPE UINT8

// Block: ID1054v1
PL_MARKING_TYPE PL_currentMarkingF[] =
{ 1, 0, 0, 0, 5, 0, 1, 0 };
PL_MARKING_TYPE PL_newMarkingF[] =
{ 1, 0, 0, 0, 5, 0, 1, 0 };

// No arc information is needed:
// Weight == Constant

// Block: ID3054v1
const UCHAR * TR_IdF[] =
{ "arriveUp", "canEnter", "arriveDown", "canLeave", "exitDown", "exitUp" };

// Block: ID3088v1
const PL_INDEX_TYPE TR_inPlaceIndexF[] =
{ 0, 2, 4, 1, 3, 7, 5, 6 };
#define TR_inPlaceIndex(trIndex)
TR_inPlaceIndexF[trIndex]

// Block: ID5452v1 // bit array
UINT8 IO_inputEventBufferF[] = { 0 };
UINT8 IO_inputEventMemoryF[] = { 0 };
```

Figure 4. A control access for a parking lot.
### 7. Conclusions and future work

The presented code generator creates code that can be used for multiple purposes, namely: execution in different and specific hardware platforms, simulation, and analysis purposes. The block generator database allows a flexible and growable technique to generate code tailored for each specific platform resources, taking into account the characteristics of each Petri net class and model. The current tool is ready to support the generation of code in other high-level languages. Especially interesting will be the generation of code in other languages, namely VHDL for hardware implementations, SystemC for system-level designs, and languages for Programmable Logic Controllers programming. Conflict solving will be given special attention. Several optimisation options can still be added and the variety of available code blocks can still be extended to allow finer degrees of optimisation. This should allow further testing regarding different scenarios for memory/speed balancing on each platform and with various model sizes. Presently, the association between data structures and block generators is hard-coded in the generator application. This too, will be made configurable by the user.

### Acknowledgments

The presentation of this work at ETFA’2005 is supported by the Portuguese FCT sponsored POSC/EIA/61364/2004 FORDESIGN project (http://www.uninova.pt/fordesign).

### References


