# Modeling, Verification and Code Generation for FPGA with Adaptive Petri Nets

Carl Mai, René Schöne, Johannes Mey, Michael Jakob, Thomas Kühn and Uwe Aßmann

Technische Universität Dresden
Dresden, Germany
Email: {carl.mai, rene.schoene, johannes.mey, michael.jakob, thomas.kuehn3, uwe.assmann} @tu-dresden.de

***Abstract*—Petri nets are a formalism used to model the behavior of systems. Modeling systems with context dependent behavior is more complex and no suitable model exists, which can be used for formal verification, graphical modeling and program synthesis. With our extension, "Adaptive Petri nets", it is possible to directly model adaptive systems while still being able to utilize their expressiveness and existing model checking tools. In this work, the utilization of Adaptive Petri nets in the context of controller synthesis for Field Programmable Gate Arrays (FPGA) is demonstrated. A full workflow from an Adaptive Petri net Model to an FPGA will evaluate the system in its usability over the three components modeling, verification and code generation.**

***Keywords–Petri nets; Reconfigurable Petri nets; Inhibitor Arcs; Analysis, Exceptions, FPGA, VHDL, Code Generation***

## I. Introduction

With Adaptive Petri Nets (APN) a framework was developed, which allows to change the behavior of a Petri net at runtime. Parts of the net can be enabled or disabled based on the number of tokens in designated places. To integrate well in the existing tool landscape of Petri nets, APN are built in a way that they can be flattened in normal Petri nets. It was proven in our work submitted for ADAPTIVE 2018 that each APN can be flattened to a Petri net with inhibitor arcs [1].

In this work, we will show multiple ways to use APN to model and synthesize a digital controller. By this, it is evaluated how APN are utilized and integrated with existing tools.

The developed APN was designed with following goals in mind:

1) **Usability:** the APN syntax should be easy to use and should require a minimal learning effort.
2) **Flattening:** an APN should be flattened into an equivalent Petri net with inhibitor arcs.
3) **Small Overhead:** flattening should not significantly increase the net in size.

With *usability* as our first goal, we hope to avoid that APN remain only a theoretical concept without practical use. For this, we developed multiple representation methods to define an APN (graphical, mathematical, composition based). Our second goal, *flattening*, is supposed to allow the reuse of existing Petri net tools. Flattening also improves the usability since an existing Petri net based project can use APN just on top without further modifications to their infrastructure. Having *small overhead* as our goal, influences decisions of the defined semantics, so a flattened APN does not explode in size and is still usable. However, there is a trade-off between usability and small overhead.

Developing a controller on an FPGA provides various challenges for a programmer. Due to its parallel and asynchronous nature, most logical controllers require some synchronization points. These synchronization points are used, e.g., to execute routines consecutively [2] or wait for sensors and actuators [3]. While the most commonly used models for this are state machines, they can handle only one state at a time. Therefore, state machines cannot be used well in systems where the state is dependent on multiple contexts [4], [5].

When modeling a system with reconfigurable behavior, e.g., reconfigurable manufacturing systems, the system not only has to handle multiple contexts, but has to adapt its behavior to contexts [6]. State machines and Petri nets fall short in this kind of scenario, since modeling of context dependent behavior cannot be directly expressed [7]. With Adaptive Petri nets (APN) [1], we proposed a Petri net extension, which adds a syntactic, semantic, and graphical extension to Petri nets to support modeling self-adaptiveness.

The remainder of the paper is structured as follows. In Section II, the related work is reviewed. It has three focus points, the extension of Petri nets to support adaptivity and the use of Petri nets for circuit synthesis as well as the intersection of both. Section III is a background chapter and will contain the formal background of Petri nets and introduces the concept of APN. Next, in Section IV, our proposed workflow from APN to circuit is described. In the end, in Section V, we will give an example of a circuit controller, which is modeled, verified, and then synthesized into a circuit with VHDL (Very High Speed Integrated Circuit Hardware Description Language) according to our workflow. Finally, an outlook and conclusion is given.

## II. Related Work

In this section, we will survey three types of related work. The first type of related work, which we present here, covers Petri nets with adaptive behavior changes. The other type of related work covers the synthesis of Petri nets to circuits or HDLs (Hardware Description Languages). And finally, we survey the related work, which is a combination of the prior types, i.e., Adaptive Petri nets synthesized to circuits or HDLs.

### A. Petri nets with changing runtime behavior

While Petri nets themselves already express runtime behavior, there is no construct to express changes in runtime behavior.

It is possible to express changing runtime behavior directly within Petri nets. However, this will model the adaptivity on the same layer as the business logic, and complicates the final designs because of an intermingling of concerns.

In the following, we review existing work concerning Petri nets, which can change their behavior at runtime.

**Object Petri nets** [8] are Petri nets with special tokens. A token can be a Petri net itself and therefore, nets can be moved inside a main net. This type of net can be used for modeling multiple agents, which move through a net representing locations. The agents change their internal state and have different interactions based on the location inside the net. This approach extends the graphical notation of Petri nets. Analysis of object Petri nets is possible with the model checker Maude [9] and by conversion to Prolog. It was not shown that object Petri nets can be flattened to standard Petri nets, though.

Reconfiguration with graph-based approaches is a topic of Padberg's group. They developed the tool **ReConNet** [10], [11] to model and simulate reconfigurable Petri nets. A reconfiguration is described as pattern matching and replacement that are evaluated at runtime. This notation is generic and powerful, but cannot be represented in the standard notation of Petri nets. It was also not a goal to flatten them into standard Petri nets. Verification is possible with Maude.

Another graph-based reconfiguration mechanism is **net rewriting systems** (NRS) [12]. The reconfiguration happens in terms of pattern matching and replacements with dynamic composition. The expressive power was shown to be Turing equivalent by implementation of a Turing machine. Additionally, an algorithm for flattening to standard Petri nets was provided for a subset of net rewriting systems called reconfigurable nets. This subset constrains NRS to only those transformations, which leave the number of places and transitions unchanged, i.e., only the flow relation can be changed. Flattening increases the size of transitions significantly, i.e., by the number of transitions multiplied by the number of reconfigurations. With **improved net rewriting systems** [13], the NRS were applied in logic controllers. The improved version of NRS constrains the rewrite rules to not invalidate important structural properties, such as liveness, reversibility, and boundedness.

**Self-modifying nets** [14] were already introduced in 1978 to permit reconfiguration at runtime. Arcs between places and transitions are annotated with a weight specifying the number of tokens required inside the place until the transition becomes enabled. To achieve reconfiguration, these weights are made dynamic by linking them to a place. The number of the weight is then determined by the number of tokens inside this referenced place. This mechanism allows the enabling and disabling of arcs and therefore, can change the control flow at runtime. However, the authors state that reachability is not decidable [14].

**Guan et al.** [15] proposed a dynamic Petri net, which creates new structures when firing transitions. To achieve this, the net is divided in a control and a presentation net. In the control net annotations on its nodes instruct the presentation net for structural modifications. Verification and reducibility were explicitly excluded by the authors.

A practical example was shown in **Bukowiec et al.** [16], who modeled a dynamic Petri net, which could exchange parts of the net are based on configuration signals. Defining reconfigurable parts was done with a formalism of hierarchical Petri nets. The dynamic parts of the nets were modeled with subnets to generate code for a partially reconfigurable Field Programmable Gate Array (FPGA). Since this work was of more practical nature, the reconfiguration and transformation were not formalized. However, it was shown by Padberg et al. [10] that this kind of net can be transformed into a representation, which can be verified using Maude.

**Dynamic Feature Petri nets (DFPN)** [17] support runtime reconfiguration by annotating the Petri net elements with propositional formulas. These elements are then enabled or disabled based on the evaluation of these formulas at runtime. The formulas contain boolean variables, which can be set dynamically from transitions of the net or statically during initialization. Their model extends the graphical notation with textual annotations. It was shown that they can be flattened to standard Petri nets [18]. Compared to Adaptive Petri nets, this type of net is problem specific and has the limitation of indirection by boolean formulas. A boolean formula cannot express numbers easily, only by encoding them in multiple boolean variables. In DFPN the net is modified by firing transitions, while in Adaptive Petri nets the net is modified by the number of tokens inside a place.

With **Context-adaptive Petri nets** [19], ontologies were combined with Petri nets to model context dependent behavior in Petri nets. These nets are included in an existing Petri net editor. By this, context-adaptive Petri nets support modeling, simulation and analysis. It is unclear whether this approach would also work on larger nets, since it was not detailed how the analysis is implemented. Additionally, the flattening of these nets is not supported.

**Hybrid Adaptive Petri nets** [20] are a Petri net extension coming from the field of biology. These nets extend non-standard Petri nets with a special firing semantic. A transition can fire discrete, which will consume and produce a single token and then wait a specified delay for the next firing. In continuous mode a transition will not have a delay. This Petri net is adaptive by switching between those two modes. Compared to our work this is out of scope since non-standard Petri nets are used and adaptivity is restricted to transitions only.

There exist two surveys, which also summarized the related work on this topic. In the work of Gomes et al. [21], the change of behavior at runtime is classified as dynamic composition. It is characterized as "*rare*", arguing that it "*radically changes the Petri net semantics* and *complicates the available analysis techniques*". A more thorough overview of the related work can be found in Padberg et al. [7].

### *B. Circuit synthesis from Petri nets*

Transforming Petri nets into circuits was already done in the 1970s [22] only a few years after the concept of Petri nets was published by Carl Adam Petri [23]. This already highlights the strong relationship between Petri nets and circuit design. The complete history, for Petri net synthesis into circuits, can be read in [5].

A noticeable trend since the 1970s until today, is the more abstract view on hardware. Especially with the introduction of HDLs like VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) and Verilog, but also with the wide availability of FPGA, the gap between theoretical designs and practical implementations diminished. Furthermore,

the supported net classes increased over time. While in the beginning only basic net classes were supported, nowadays exist synthesis algorithms for high level nets, too. It can be observed that the interest in Petri nets as a design aid for digital systems has increased [5]. This is attributed to two reasons. Petri nets naturally capture the relations, concurrency and conflicts of digital systems. Additionally, Petri nets are very simple but expressive and formally founded [5, p.4]. The development of FPGA can be seen as the main contributor in this field. On one hand, it can be used to rapidly prototype and test algorithms and technologies, on the other hand, it is an easier to reach target for synthesized circuits. With this, the technological stack for implementing new tools around Petri net matured in the last decade much faster than before.

Several surveys were done in this field. A very early overview was given by Agerwala in 1979, where it was mentioned as part of a survey for practical Petri net applications [24]. The survey focused on the synthesis algorithms by Dennis' group in the 70s [22]. For Finite State Machine (FSM) an in-depth survey was done by Moore and Gupta [25]. Not only use-cases and approaches were surveyed, also Petri net types and analysis methods. A more practical article on FSM implementation in Verilog was written by [26]. In 1998, two more surveys were published, the survey from Yakovlev and Koelmans [4] and from Marranghello [5] concerning asynchronous and synchronous synthesis of embedded controller, respectively. After that only very small surveying was done, as part of related work in [27], [28], [29], [30].

Petri net synthesis can be classified into three general classes: type of implementation, type of encoding and type of Petri net.

The separation between **synchronous and asynchronous implementation** is the already the focus of two surveys from 1998 [5], [4]. The decision is largely dependent on the use-case. The **type of encoding** is well elaborated in [5] and similarly in [4]. There exist three different types. *Direct encoding* also called *one-hot encoding* or *isomorphic places encoding* [31], is the 1:1 mapping of Petri net places into a circuit element (e.g., a flip-flop). This encoding guarantees a circuit and has the shortest synthesis time, as there are no complex calculations involved. A disadvantage is the higher number of flip-flops required. To tackle the problem, *logical encoding* gives each state or transition in a (sequential) Petri net a code and represent this state in the circuit by complex logic. Depending on the encoding, this is either named *place-based* or *transition-based* encoding. The state-space explosion problem [32] might result in a failed synthesis. To mitigate this problem, the net can be partitioned in multiple subnets with macro nets [33] or by using Binary Decision Diagrams (BDD) for a more efficient representation [34]. The last encoding method is by building a specialized hardware, which takes a Petri net and computes the firing. This solution is the most space efficient for large nets and allows the highest grade of reconfiguration. Disadvantages are their higher initial effort and slower execution speed [27], [35].

Overlooked in the previous taxonomies, the **Petri net type** is also a distinctive characterization. Most implementation work on safe Petri nets, some on k-bounded and some with colored tokens. Additionally, often the non-deterministic feature of Petri nets is not supported when synthesizing to circuits or needs special care [36].

### C. Circuit synthesis from Petri nets with changing runtime behavior

While the synthesis of Petri nets into circuits is researched for a long time, only in recent years the research focuses also on Petri nets with changing runtime behavior. The first work, looking at this topic is [16]. Here, a non-formal Petri net model, which allows reconfiguration at runtime, is synthesized into VHDL for a partial reconfigurable FPGA. Relatively similar is [37], in which a state machine is synthesized for a partial reconfigurable FPGA. Both approaches switch the runtime behavior based on the context. The use-cases are based on an industrial and smart home scenario, respectively.

Similar research is also performed outside of FPGA synthesis. In [38] the ReConNet of Padberg et al. [10] is utilized to synthesize a reconfigurable manufacturing system (RMS). Here, a formal approach was used to model the system, verify the Petri net properties and then synthesize the RMS.

## III. Preliminaries

This section defines the preliminaries, used in this work. The mathematical notation of Petri nets is explained in this section together with some properties, which can be verified with model checking. The concept of Adaptive Petri nets is explained together with an algorithm to flatten APN to Petri nets with inhibitor arcs.

### A. Petri net definitions

*Definition 1:* A **Petri net** [32] is a directed, bipartite graph and can be defined as a tuple $\Sigma = (P, T, F, W, M_0)$. The two sets of nodes are $P$ for places and $T$ for transitions, where $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$. $F$ is a set of arcs, describing the flow relation with $F \subseteq (P \times T) \cup (T \times P)$. $W : F \to \mathbb{N}$ is a weight function. $M_0 : P \to \mathbb{N}$ is the start marking.

Referencing an element of the tuple is done in dot notation: for a Petri net $\Sigma$, we reference the places $P$ by $\Sigma.P$.

*Definition 2:* For an element $x \in P \cup T$, $\bullet x = \{y | (y, x) \in F\}$ and $x\bullet = \{y | (x, y) \in F\}$.

E.g., $t\bullet$ with $t \in T$ refers to the set of places, which are connected with an arc originating from $t$. We call those preset and postset, respectively.

*Definition 3:* A **marking** is defined as a function $M : P \to \mathbb{N}$.

A Petri net is a static model, in which only the marking changes. $M_0$ is the start marking. After firing a transition, the marking changes.

*Definition 4:* A transition $t \in T$ is **enabled** if all places $p \in \bullet t$ have a marking of at least $W(p, t)$ tokens, where $W(p, t)$ is the weight for the arc between $p$ and $t$.

*Definition 5:* Iff a transition $t$ is enabled, it can **fire** and the marking of each $p \in t\bullet$ is incremented by $W(t, p)$ and the marking of each $p \in \bullet t$ is decremented by $W(p, t)$.

*Definition 6:* If there exists a $k \in \mathbb{N}$ for a $p \in P$ such that, starting from $M_0$, every reachable marking $M(p) \leq k$, we speak of $p$ as **k-bounded**.

A bounded place never contains more than $k$ tokens. If $k$ equals 1, this place is called **safe**.

### B. Inhibitor arcs

To model the negation inside Petri nets, e.g., "fire this transition only when less than x tokens are inside this place", inhibitor arcs can be used. With inhibitor arcs, the flow relation of Petri nets is extended with an arc, which disables a transition when the connected place has more than a specified number of tokens in it. A Petri net with inhibitor arcs can implement a Turing machine [39], while this is not possible with standard Petri nets. Because of the change of expressiveness, the available tools for model checking are reduced, for example, the halting problem cannot be solved in general for Turing complete languages.

*Definition 7:* An **Inhibitor Petri net** is a tuple $\Sigma = (P, T, F, I, W_I, W, M_0)$. With the same defintions as previously mentioned. Additionally this Petri net contains the set of **inhibitor arcs** $I : (P \times T)$ and a weight $W_I : I \to \mathbb{N}$

To simplify notation, we define the inhibiting set of a transition $t$ as $\circ t = \{(p, t) \in I\}$.

*Definition 8:* A transition $t$ is **enabled**$_i$, iff all places connected by an inhibitor arc are below the weight $M(p) < W_I(p, t)$ for all $p \in \circ t$ and the transition is enabled as defined in Def. 4.

*1) Flattening to a Petri net without inhibitor arcs:* In general, a Petri net with inhibitor arcs is Turing complete. When a place with an inhibitor arc is bounded, the inhibitor arc can be replaced with a semantic preserving structure without an inhibitor arc [40].

### C. Graphical notation

Places are drawn as circles: ◯, their marking is drawn as black dots ⊙. Transitions are drawn as black rectangles (horizontal or vertical) ▮. The flow relation is drawn with directed arcs between places and transitions ⟶. Inhibitor arcs are only drawn from places to transitions and get a circle head: —○.

### D. Properties and analysis of Petri nets

Petri nets support various ways to verify its properties. The most commonly used analysis techniques check for reachability, boundedness, deadlocks and liveness [32]. With these properties, it is possible to verify the correctness of the model according to its specifications. In this section we will first describe these properties and then two tools used for analysis.

The basis for most model checking techniques in Petri nets is **reachability**. This technique answers the question, whether there exists a firing sequence to get from a marking $M_1$ to $M_2$. There exists also the sub-marking reachability, which ignores the marking of some places [32]. Besides for Petri nets with inhibitor arcs, the reachability is decidable but requires exponential space and time [41].

**Boundedness** is used to determine, whether the marking of a particular place is such that the number of tokens is always lower than a k with $k \in \mathbb{N}$ (see Def. 6). Boundedness is very important for synthesis of Petri nets to guarantee that no buffer will overflow.

The property **liveness** refers to a Petri net, in which, starting with any marking $M_0$, there exists a firing sequence such that all transitions can be fired. This is a very strong property, which can be checked on five different levels (L0-L4), where each level adds some relaxation [32].

A **deadlock** in the context of Petri net refers to a marking, in which no transition can fire [32].

For analyzing Petri nets, several tools exist. Here, shortly two tools are explained. For low level analysis and especially for checking reachability, we chose LoLA (Low Level Analyzer) [42], [43] as it is multiple times the winner in the Petri net model checking contest in the category of reachability [44]. To check for reachability, LoLA accepts formulas in either temporal logic (either linear temporal logic (LTL) or computation tree logic* (CTL*)). LoLA automatically uses the fastest temporal logic for a given query, therefore, we will draw no distinction here. LTL and CTL* both build on top of the propositional calculus and extends it with temporal quantors. Such quantors are X for next state, G for global state, F for finally (e.g., a state will be reached in a finite number of steps). One of the design goals of LoLA is to keep the architecture relatively clean. That is why, only the basic type of Petri nets with no inhibitor arcs or colored tokens is supported.

The second tool, regarded in this work is Tina [45]. This tool is much more high level than LoLA. It comes bundled with a graphical editor and simulator, it can convert many different Petri net formats and has an interpreter, which can check for most basic properties like liveness, deadlocks and boundedness. However, the interpreter is much slower than LoLA.

### E. Adaptive Petri nets

Adaptive Petri nets (APN) extend Petri nets with a concept to change the behavior of the net at runtime. This is done by defining one or more *configuration points*, which in turn consist of a set of nodes, which are configured and a place (*configuration place*) together with a marking, which enables or disables the set of nodes. When the set of nodes is enabled, the behavior of the Petri net is not changed. When the set of nodes is disabled, no new tokens can be emitted from outside into the set of nodes.

Following this informal description, the definition and semantics is given here.

*Definition 9:* An APN is a tuple $\Sigma = (P, T, F, W, M_0, C)$, based on Petri nets of Def. 1, with $C = \{c_1, c_2, \ldots\}$ as the set of configuration points.

*Definition 10:* A **configuration point** is a tuple $c = (p, w, N, E)$ referencing the nodes of a containing Petri net $\Sigma$.

- $p \in \Sigma.P$, a place that we will call *configuration place*.
- $w : \mathbb{Z} \setminus \{0\}$, a weight
- $N \subseteq (\Sigma.P \cup \Sigma.T)$, the nodes that are configured
- $E \subseteq N$ the external nodes of the configured net, which are reachable, even if the configured net is disabled

*Definition 11:* The set of **external nodes** ($E \subseteq N$) are nodes of $N$ which are connected to nodes outside of N. Usually defined like this - but a custom definition is possible, too: $E = N \cap \{x | (x \in \bullet n \cup x \in n\bullet) \forall n \in ((P \cup T) \setminus N)\}$

*Definition 12:* The set of **internal nodes** for a configuration point is calculated by $G = N \setminus E$.

With these definitions, the structural part of APN is described. — In the next definitions, the runtime semantics of APN are described.

*Definition 13:* A configuration point $c \in C$ is **enabled**, iff $(c.w > 0 \wedge M(c.p) \geq c.w) \vee (c.w < 0 \wedge M(c.p) < |c.w|)$.

**Algorithm 1** Flattening of an Adaptive Petri net

```
1: procedure FLATTEN((P, T, F, W, M, C, I))
2:     for ∀c ∈ C do
3:         for ∀p ∈ c.E ∩ P do
4:             for ∀t ∈ p• ∩ c.G do
5:                 ConnectByArc((⊤, c, t, F, I, W))
6:             end for
7:         end for
8:         for ∀t ∈ c.E ∩ T do
9:             if (t• ∩ c.N ≠ ∅) ∨ (•t ∩ c.E ≠ ∅) then
10:                t_2 ← Duplicate(t, P, T, F, W, C, I, W_I)
11:                F ← F \ ((t_2 × c.N) ∪ (c.E × t_2))
12:                ConnectByArc((⊤, c, t, F, W, I, W_I))
13:                ConnectByArc((⊥, c, t_2, F, W, I, W_I))
14:            end if
15:        end for
16:        C ← C \ {c}
17:    end for
18: end procedure
```

With *M* being the marking function of Def. 3. As a shorthand, the set of enabled configuration points is defined as $C_e \subseteq C$.

An enabled APN is not changing the behavior of the Petri net. A disabled APN stops the flow of tokens from $E$ to $N$. By this, the definition of fire Def. 5 must be modified as well as the definition of enabling Def. 4. These modifications are defined in Defs. 16 and 17, respectively.

*Definition 14:*
- The set of configuration points a node belongs to is defined by the function $B^N : (P \cup T) \rightarrow \mathbb{P}(C)$ with $B^N(n) = \{c | c \in C \wedge n \in c.N\}$.
- The set of configuration points, in which a node is external, is defined by the function: $B^E : (P \cup T) \rightarrow \mathbb{P}(C)$ with $B^E(n) = \{c | c \in C \wedge n \in c.E\}$.
- The set of configuration points, in which a node is internal, is defined by the function: $B^G : (P \cup T) \rightarrow \mathbb{P}(C)$ with $B^G(n) = \{c | c \in C \wedge n \in c.G\}$.

*Definition 15:* The *configured postset* and *configured preset* of a transition $t$ is defined as $t\bullet_c = t\bullet \setminus \{p | c \in (B^E(t) \setminus C_e) \wedge p \in c.N\}$ and $\bullet_c t = \bullet t \setminus \{p | c \in (B^E(t) \setminus C_e) \wedge p \in c.E\}$, respectively.

*Definition 16:* Iff a transition $t$ with $B^E(t) \neq \emptyset$ is enabled, it can **fire**$_a$ and the marking of each $p \in t\bullet_c$ is incremented by $W(t,p)$ and the marking of each $p \in \bullet_c t$ is decremented by $W(p,t)$. The fire semantics of all other transitions are following Def. 5.

*Definition 17:* A transition $t \in T$ is **enabled**$_a$, iff it is enabled according to Def. 4 and the following condition holds true $\{p | p \in \bullet t \wedge p \in c.E;\ \forall c \in (B^G(t) \setminus C_e)\} = \emptyset$.

For a disabled configuration point, the movement of tokens from $E$ to $N$ is prohibited in Def. 17 for transitions in $N$. The movement of tokens to places in $N$ is prohibited with Def. 16.

An APN can be flattened to a Petri net with inhibitor arcs [1]. Furthermore, it was shown that in some cases no inhibitor arcs are created and that, with the algorithm of [40], an inhibitor arc from a k-bounded places can be flattened to a Petri net without inhibitor arcs. When the place is 1-bounded, the overhead is minimal with just one additional place.

**Algorithm 2** Helper method to enable or disable a transition by a configuration place

```
1: procedure CONNECTBYARC((e, c, t, F, I, W, W_I))
2:     if ((c.w > 0) ∧ (e = ⊤)) ∨ ((c.w < 0) ∧ (e = ⊥)) then
3:         if (c.p, t) ∈ F ∨ (t, c.p) ∈ F then
4:             if (t, c.p) ∈ F then
5:                 F ← F ∪ {(c.p, t)}
6:                 W(c.p, t) ← |c.w|
7:                 W(t, c.p) ← |c.w| + W(c.p, t)
8:             end if
9:         else
10:            F ← F ∪ {(c.p, t), (t, c.p)}
11:            W(c.p, t) ← |c.w|
12:            W(t, c.p) ← |c.w|
13:        end if
14:    else
15:        if (c.p, t) ∈ I then
16:            if W_I(c.p, t) > |c.w| then
17:                W_I(c.p, t) ← |c.w|
18:            end if
19:        else
20:            I ← I ∪ {(c.p, t)}
21:            W_I(c.p, t) ← |c.w|
22:        end if
23:    end if
24: end procedure
```

**Algorithm 3** Helper method to duplicate a transition

```
1: procedure DUPLICATE((t, P, T, F, W, C, I, W_I))
2:     T ← T ∪ {t_2} with t_2 ∉ (P ∪ T)
3:     F ← F ∪ {(t_2, p) | p ∈ P ∧ (t, p) ∈ F}
4:     F ← F ∪ {(p, t_2) | p ∈ P ∧ (p, t) ∈ F}
5:     I ← I ∪ {(p, t_2) | p ∈ P ∧ (p, t) ∈ I}
6:     W ← W ∪ {(t_2, p) | p ∈ P ∧ (t, p) ∈ W}
7:     W ← W ∪ {(p, t_2) | p ∈ P ∧ (p, t) ∈ W}
8:     W_I ← W_I ∪ {(p, t_2) | p ∈ P ∧ (p, t) ∈ W_I}
9:     for ∀c ∈ C do
10:        if t ∈ c.N then
11:            c.N ← c.N ∪ {t_2}
12:        end if
13:        if t ∈ c.E then
14:            c.E ← c.E ∪ {t_2}
15:        end if
16:    end for
17: end procedure
```

*1) Multiple configuration points:* When multiple configuration points are configuring a set of nodes, the intersection of internal nodes of these configuration points are only enabled, when all configuration points are enabled. Therefore, the logical operator *and* is represented with the combination of multiple configuration points.

### F. Scalability of the flattening approach

We argue that one of its strengths of Adaptive Petri nets is the ability to flatten it and then utilize existing model checking tools. This will only be possible, when the flattening itself will not increase the state-space of the resulting net exponentially, such that the model checking can not work in reasonable time

for larger nets.

To clarify the three stages of flattening we perform, the type of Petri net is marked in the sub-script of the set. I.e, places of an APN are denoted as $P_{APN}$, places of a Petri net with inhibitor arcs are denoted as $P_{inh}$, and places of a Petri net without inhibitor arcs are denoted as $P_{p/t-net}$. The same syntax is also used for transitions.

The worst-case increase of places, when flattening from an APN to a Petri net with inhibitor arcs is: $|P_{APN}| \in o(|P_{inh}|)$, the number of places does not increase when flattening an APN. When flattening a safe (1-bounded) Petri net with inhibitor arcs, it is $2 \cdot |P_{inh}| \in o(|P_{p/t-net}|)$. Calculating the worst-case increase for the number of transitions might be misleading, as it hardly reflects reality since it will assume many overlapping subnets: $2^{|C|} \cdot |T_{APN}| \in o(|T_{inh}|)$. With each configuration point the number of transitions can double. When flattening a safe Petri net with inhibitor arcs, the amount of transitions does not increase.

For model checking tools, the most critical criteria to solve a net, is the size of the state-space. The state-space in turn is heavily influenced by the number of places a net contains. For a safe Petri net, the state space is in the worst case $o(2^{|P|})$.

With Adaptive Petri nets, the size of places does not increase when flattening to a Petri net with inhibitor arcs. Although, the size of transitions can increase exponentially to the number of configuration points. For scalability, the most limiting factor is the flattening of inhibitor arcs, which can result in an exponential amount of additional places and transitions. Since the semantics of the APN is just boolean (enabled or disabled), users should be able to model the net in a way, that all configuration places are 1-bounded. This will only add one additional place per configuration place. From practical experience, we never found the model checking as our limiting factor.

*1) Improvements to previously published work:* After the publication in [1], some improvements were found. To better compare these works, we will list the changes here.

Internal nodes of $C$ were defined as $I$, which was ambiguous to the set of inhibitor arcs. Now the symbol $G$ is used.

The set of external nodes $E$ was previously set implicitly with a formula. Now it is part of the definition of an APN. This must be done to support commutativity in evaluation and flattening, when combining multiple configuration points over an intersecting subnet. This change can be especially noticed in Algorithm 3 and Algorithm 2.

## IV. Workflow from Adaptive Petri nets to FPGA

Modeling a circuit with an FSM or Petri net has many advantages, already described in Section II. We propose an architecture, which generates valid VHDL code from an APN. The whole workflow is depicted in Figure 1, described later in this chapter, and finally evaluated with a practical example of a coffee machine in Section V.

In Figure 1, the transformation chain is depicted. It can be read from the left (input) to the right (output). Circles and ovals depict artifacts, e.g., files, while arcs and rectangles are transformations and computations.

### A. Input: Petri net

The transformation chain is started with various inputs. The only mandatory input is a Petri net, named *Base PN*. This Petri net can already be an APN or contain inhibitor arcs. To help with separation of concerns, a *composition* system can be used to separate the configured nodes from the base net (we employ name-based composition and net addition rules [46], [47]. For this, a *Composition Specification* may be required to describe how the multiple parts are combined. For the compositional approach, the base net contains the core functionality, which is enhanced by several features, named *PN Feature*. This concept is similar to feature-oriented programming [48].

### B. Input: Context net

The input *context net* is specified by the developer and used to separate concerns. While it is not strictly necessary for APN, we found that a separate handling of the configuration points helps when designing the nets. On one hand, it is used for separating the context information from the base net. On the other hand, it can be constructed in a way to guarantee that all places of this net are 1-bounded, simplifying the flattening of inhibitor arcs.

For the context net, three options were investigated. First a simple Petri net, which does not provide a lot of abstractions. The second investigated model is the context Petri net [49]. Context Petri nets are first described by a domain specific language (DSL), which is setting multiple contexts in relationship to each other. A relationship can be exclusion, inclusion, implication, etc. This DSL is then transformed in a Petri net, which handles the activation and deactivation based on the relationship. E.g., when a context is activated, which is in an exclusion relationship with another context, the other context is then deactivated. The third option is a modified state machine. A state machine is defined as a 4-tuple $STM = (Q, s, \Sigma, f)$. With $Q$ as a finite set of states, $s$ as the starting state, $\Sigma$ a finite input alphabet and $f : S \times \Sigma \rightarrow S$ the state transition function. For our use-case, we also add the set of contexts $C$ to the state machine. Each state can have a subset of $C$ assigned to it. An example can be seen in Figure 3. Such a state machine has the advantage that it is very concise but still can be transformed to a Petri net with little overhead by transforming all states $Q$ and events $\Sigma$ into places, all state transition functions $f$ into transitions connecting the input and output state-places correspondingly and also adding the event-place as input.

### C. Composition

Utilizing a composition system gives two advantages. It enables us to use a context net in the first place. Furthermore, it can be used to simplify the definition of configuration points, when each composed net is interpreted as the set N, the *configured nodes*, while the composed nodes are the *external nodes*.

For composition, two systems were used. A rather pragmatic approach, based on node fusion [50] via name-unification. All nets that have to be composed are put into one large net. Then all places with the same name are fused together, performing an addition of their tokens and merging the input and output arcs. Similarly, this is done for transitions. As a wildcard, the "*"-character can be used at the beginning or end of a string, which then merges with all prefixes and suffixes of that string depending on the position of this character.

The other utilized system is based on net additions [46]. Net additions consist of a DSL, in which the names of the composed Petri nets are first listed, followed by a list of node fusion sets. A node fusion set is either a set of places or a set of transitions from any composed Petri net, referenced by their name and with a new name. For example, the node fusion set $(a/b/c \rightarrow d)$ is merging the nodes $a$, $b$ and $c$ to a node named $d$. We extended net additions, to specify a configuration place together with the marking next to the name of a Petri net, such that the Petri net becomes the set $N$ of the configuration point [47].

### D. Adaptive Petri net and flattening

After the composition step, an APN is the result. Either because the *base net* was already an APN, or because the composition added configuration points to the net. The APN is then *flattened* with the algorithm of [1] to a *Petri net with inhibitor arcs*.

### E. Model checking

To utilize existing model checking tools, the inhibitor arcs can be *flattened* when the source node of this arc is bounded [51]. Model checking can be performed on user-provided rules (*Model Check: custom* with *LTL/CTL* Formulas*) and with generic rules, like deadlock detection, unreachability, unboundedness and invariants (*Model Check: generic*). These generic checks can also be used, to *Optimize* the circuit model. For example, to eliminate dead code or remove redundant places from the invariant analysis. All model checking results can be inspected by the developer to fix bugs and inconsistencies in the modeled Petri nets (*Check Results*).

### F. Circuit model

The flattened APN, a Petri net with inhibitor arcs, is transformed in a *Circuit Model*. Our circuit model consists of: connections, basic gates like AND and OR, as well as a counter. Currently not implemented is the step from Adaptive Petri nets to the circuit model. It is planned to use the dynamic reconfiguration capabilities of FPGAs for this [16], [52].

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity place is
5      generic (max: integer := 1; def: integer := 0);
6      -- I=Increment, D=Decrement, O=Out
7      port (I: in std_logic; D: in std_logic; O: out std_logic
           ; clk: in std_logic);
8  end place;
9  architecture dataflow of place is
10     signal memory: integer range 0 to max := def;
11 begin
12     process(clk)
13     begin
14         if rising_edge(clk) then
15             if D = '1' then memory <= memory - 1;
16             elsif I = '1' then memory <= memory + 1;
17             end if;
18         end if;
19     end process;
20     O <= '1' when memory > 0 else '0';
21 end dataflow;
```

Listing 1. VHDL of a place

Here, we use the following Petri net synthesis class: (see Section II): *synchronous*, *one-hot* encoded, with *k-bounded* places, *1-bounded* arcs, *inhibitor* arcs and without indeterministic constructs. Most of these restrictions are purely for pragmatic reasons, to keep the transformation to the circuit model simple. In the future, it is planned to extend the synthesis to asynchronous circuits and to support k-bounded arcs. The most important transformations can be seen in Figure 2. The transformation is modeled closely to [53]. Each place and each transition gets a one-to-one mapping in the circuit. Inhibitor arcs are represented with the logical *not*.

### G. HDL-code generation

After the circuit model was optimized, an *HDL (Hardware Description Language) Model* is generated. This model is an abstract representation of the textual VHDL code. The VHDL implementation of a place can be seen in Listing 1. From this, two HDL files are generated. One implementation file, which contains all the logic to run the Petri net and an *HDL Skeleton* is generated, which the developer can use to

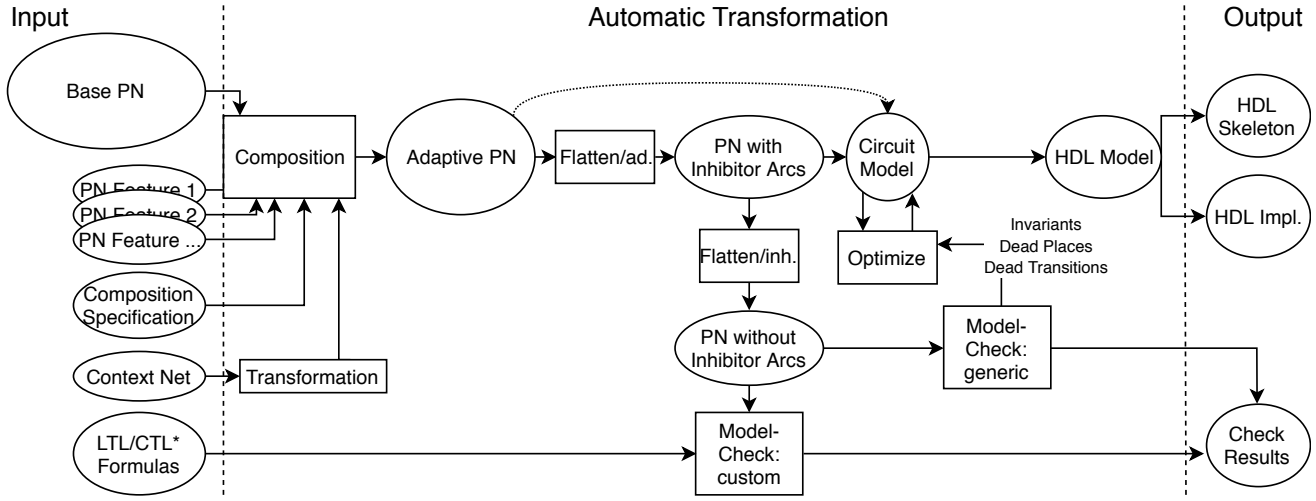


Figure 1. Transformation workflow. PN = Petri net. Ovals are artifacts, rectangles are processes.

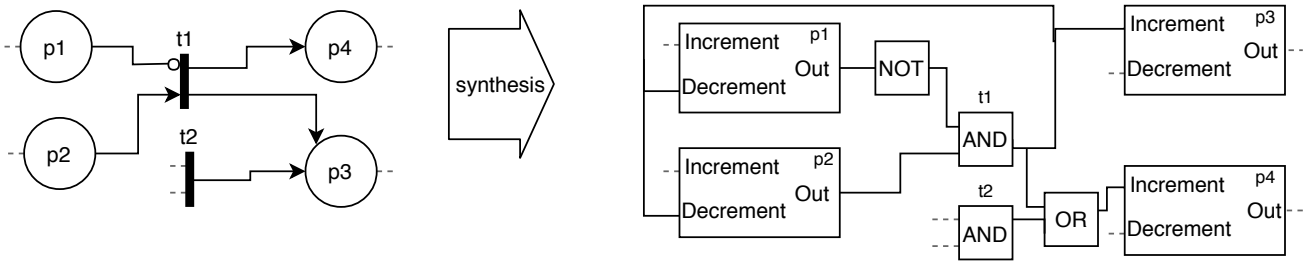


Figure 2. Petri net circuit synthesis with *one-hot encoding*

implement their functionality. The skeleton consists of the Petri net implementation and an API, exposing all important places and transitions. Unimportant nodes are those, which were created automatically. Internally this is done by prefixing the nodes with a special keyword. For implementation, transitions can be used for influencing the Petri net execution by either blocking or continuing the net-flow. Places can be used as impulses for the VHDL program to trigger the execution or directly power an actuator of the circuit, e.g., a place will start a small engine or letting an LED light blink.

## V. Synthesizing an Adaptive Petri net to an FPGA

The general workflow, described in the previous section, will be demonstrated with a realistic use-case of a coffee machine. While the example is simple enough to understand, it also demonstrates most aspects of the workflow to show how the synthesis can be extended for more complex designs.

### A. Use-case description: Coffee machine

The behavior of the coffee machine can be described in two phases: a *configuration phase*, which awaits user-input for the type of coffee they want and a *running phase*, where the machine will prepare and dispense the coffee. During the configuration phase, the configuration places are set. When the configuration phase is finished by pressing the start button, the runtime phase starts and executes the coffee machine adapted to the configuration.

Regarding the workflow of Figure 1, the input consists of a context net, a base Petri net and LTL/CTL* formulas. The composition specification is done implicitly, by composing the nodes with a unification of the names (nodes with the same name are merged, while a * will match anything).

The coffee machine consists of 3 models: the Petri net model, the context model, and the APN model. All three models are created separately. The APN model and Petri net model are only separated because of the current technical limitation that APN cannot be represented within PNML. The separation of the context model is not required but gives a nicer overall architecture as described also in Section IV-B.

The coffee machine itself operates in two phases: (I) beverage selection; (II) beverage dispensing. In Phase I, the customer can select from 5 buttons: Coffee, Cappuccino, Milk, Espresso, Start. Except for the start button, each selection will fill the place with the same name as the button with one token. This place is then used as the context configuration. Phase II can be reached, when the customer presses the start button. In this phase, the buttons are disabled and the machine starts dispensing according to the previous selection. The internal processes of the machine are controlled by the Petri net. Utilizing Adaptive Petri nets, only those parts are activated which are defined by the contexts. The Petri net can be seen in Figure 4.

### B. Modeling

The beverage selection is done with a state machine, as it can be used to represent the selection logic in a simplified and extendable way. This state machine is modeled in our STN (state transition net) notation. It consists of states (circles), events (arcs) contexts (rectangles) and a start state. In Figure 3, the state machine can be seen.

The state machine starts in the *None* state and will move to the next state when the coffee or espresso event is triggered. It will then move to the *Coffee* or *Espresso* state, respectively. The state machine is converted into a Petri net with a simple conversion algorithm, which converts STN states to Petri net places prefixed with *state_*, events to places with the *event_* prefix and STN contexts to Petri net places without a prefix. Finally, all arcs between states are converted to a transition with the previous state and event as input and the next state as output. When transforming the example of Figure 3 into a Petri net, it results in 20 transitions and 12 places.

The Petri net model can be read from different formats. Notably PNML (Petri net Markup Language), which is a standard most Petri net tools support.

The coffee machine is modeled with the Petri net of Figure 4. This net already integrates the state machine from Section IV-B with the places *event_**, *state_None*, *Coffee*, *Espresso* and *Milk*. The net starts with a token inside place *stopped*, which allows to trigger the transitions starting with *req.*. The *-sign matches *req.Coffee*, *req.Espresso* and *req.Milk*. The net continues with the transition *ingredients*, if no token is inside any *event_** place and no token in *state_None*. This is required, so that our state machine is not in an intermediate state with unprocessed

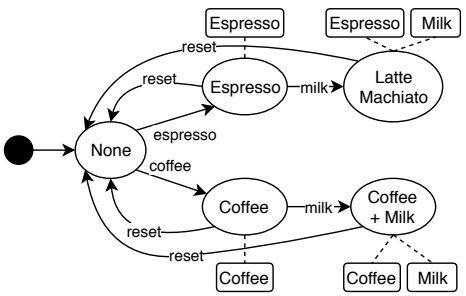


Figure 3. State machine for the selection inside the coffee machine

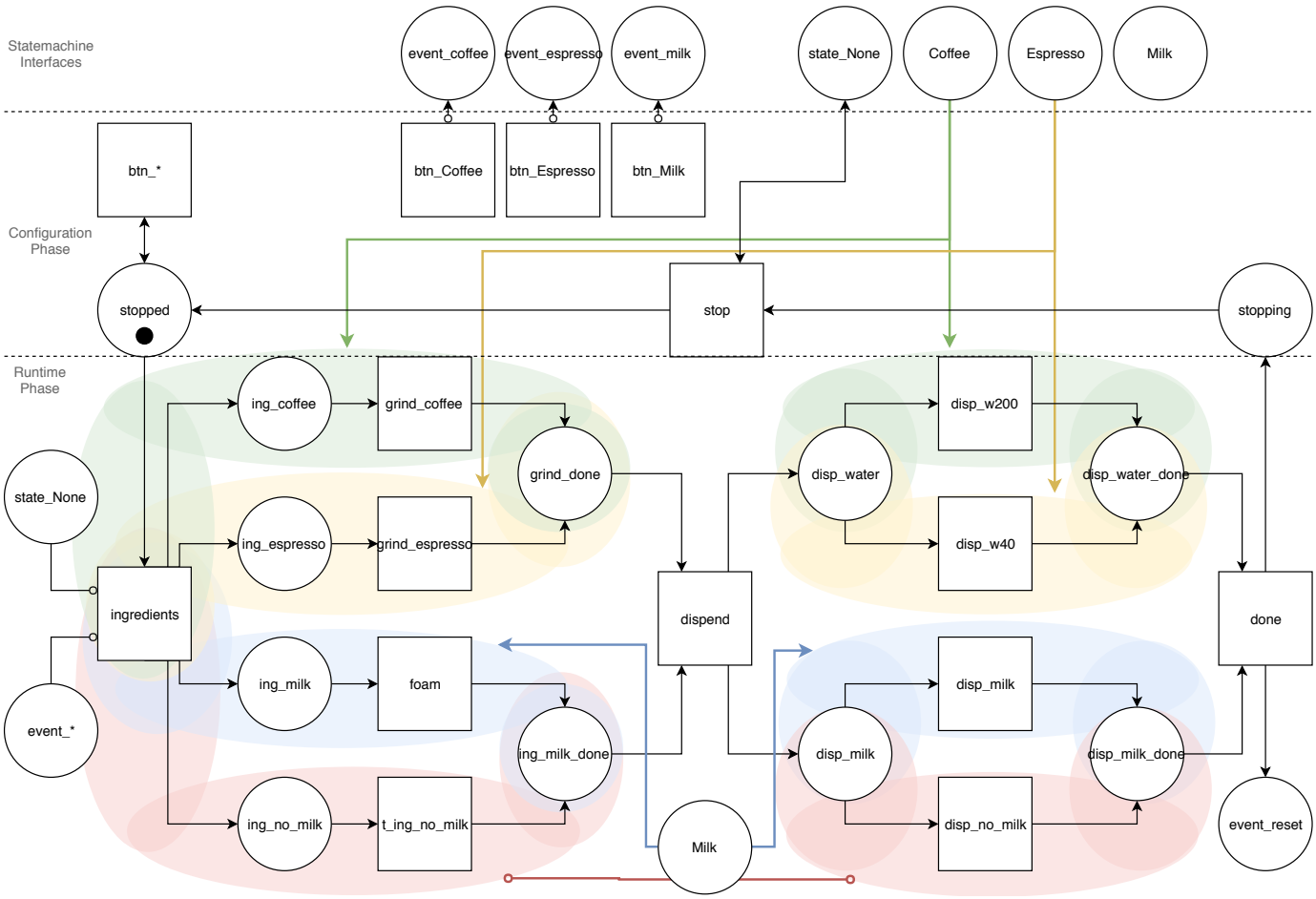


Figure 4. Petri net for the coffee machine with 4 configuration points. $C_1$={Coffee, 1, {ing_coffee, grind_coffee, disp_w200},{ingredients, grind_done, disp_water, disp_water_done}$C_2$={Espresso, 1, {ing_espresso, grind_espresso, disp_w40},{ingredients, grind_done, disp_water, disp_water_done}$C_3$={Milk, 1, {ing_milk, foam, disp_milk},{ingredients, ing_milk_done, disp_milk, disp_milk_done}$C_4$={Milk, -1, {ing_no_milk, t_ing_no_milk, disp_no_milk},{ingredients, ing_milk_done, disp_milk, disp_milk_done}

events and is also not in the *None*-state. After that, a token is put into all following places, representing ingredients for coffee, espresso and milk. The places and transitions will converge into the *dispend* transition in the middle of the figure. The ingredients will be later annotated with APN-structures. Similarly the subnet between *dispend* and *done* dispenses water and milk according to the specification and configured by the APN-structures. The coffee making process is finishing with the *done*-transition, which creates a token in *event_reset* to reset the state machine on the *None*-state and a token inside *Stopping*. The initial *stopped* state is reached, when the *stop*-transition fires, which only happens when a token is inside *state_None*.

### C. Flattening

The resulting composed Adaptive Petri net consists of 35 transitions and 24 places with 4 configuration points. When this net is then flattened to a Petri net with inhibitor arcs, the size increases to 50 transitions and 24 places. The number of transitions increases a lot, because the transition *ingredients* is an incoming external node in four configuration points. This requires to duplicate this transition $2^4$ times. However, the flattening algorithm is not yet optimizing the duplication. It will not prune illegal configurations (e.g., espresso and coffee can never be selected simultaneously).

When size is critical, the designer should watch out that the incoming external nodes are not transitions like it is done with the configuration points at the *disp* step (i.e., disp_water and disp_milk). Here, no new transitions or places are added to the net.

After the APN is flattened, it is a Petri net with inhibitor arcs. This can be further flattened to remove all inhibitor arcs with the algorithm of [40]. As prerequisite for flattening inhibitor arcs, the place connected to the inhibitor arc must have a known, finite bound. We know from all places that they are 1-bounded because they are the result of our state machine. After flattening, the net contains 58 transitions and 32 places, an increase of 8 transitions and 8 places.

### D. Model checking

A flattened Petri net can be model checked. We utilize Tina, which we chose because it has good support for PNML, can convert it to other formats, has a graphical editor, and checks the net for basic properties in a well readable format. Additionally, we utilize LoLA (Low Level Analyzer), which is winner in several model checking competitions and allows to build complex LTL/CTL* formulas [43]. There are two kinds

of checks performed: automatically generated and manual tests. We will not describe all tests, but instead give two examples of each category. For automatically generated tests, we classify theses tests into those for user feedback and those for net optimization. Checks for User feedback is testing the net for reversibility, boundedness, and deadlock freeness. Those are all checked by default in Tina. In LoLA, the deadlocks are checked by *EF DEADLOCK*. Checks for optimizations are searching for invariants. In LoLA such a check would look like this: *AG((A = 1 AND (B = 1)) OR (NOT(A = 1) AND NOT(B = 1)))* with *A* and *B* being places. The quantifier *AG* modifies the temporal predicate that this formula is only true, if all states within the state-graph of the net conform to this rule.

The coffee machine net has 4 invariants, which are all inside the state machine, e.g., Coffee = State_Coffee OR State_Coffee_Milk. With an invariant, not every place needs to be represented with a memory, but can be represented with a logical expression instead.

Besides the automatic formulas, the user can also specify manually what is of interest to him, which requires domain knowledge. In the following, two manually specified rules:

- LoLA rule: *AGEF(Coffee = 1 AND Running = 1 AND AF( Grind_Coffee = 1))* — when Coffee is selected, the grind_coffee place is always selected afterwards.
- LoLA rule: *AGEF(Coffee = 1 AND Milk = 0 AND Running = 1 AND NOT AF(Milk_Heating = 1))* — when Coffee is selected, Milk is always deselected, place Running contains a token, and we will not reach the place Milk_Heating.

### E. Generation of VHDL code

Based on the flattened Adaptive Petri net, the workflow will also create a coarse circuit model. This circuit model is generated with the transformation described in Figure 2. Here, each place is transformed into a counter and each transition in a logical AND. Currently, only synchronous circuits are generated, but there exist implementations, which do not need a clock and therefore, work asynchroneously. This coarse circuit model is then optimized to minimize the number of connections and gates. Furthermore, the optimization step receives input from the model checker, to remove invariants and dead nodes.

From the coarse circuit model, an abstract representation of the VHDL code is generated, which is transformed to actual source code in a last step. The source code is divided in two parts: an implementation part, which contains all the logic to run the Petri net and a skeleton, which contains the interface places and transitions as signal declarations. The skeleton can be later utilized by the programmer to implement additional logic. The resulting skeleton is 150 lines (3 lines for each place and 2 lines for each transition). The resulting Petri net implementation code has a length of 280 lines. Within the skeleton, the engineer has write access to the setter of all places, read access to the boolean output of all places, and write access to all transitions, where a low-signal can stop the transition from firing.

A small example of both files is given in Listing 2 and Listing 3, which consists of a single place connected to a transition. While the implementation itself must not be understood by the developer, it is still printed in a readable format. The skeleton must only be changed beginning on Line 20 for runtime behavior.

```
1  library IEEE;
2  use std.textio.all;
3  use IEEE.STD_LOGIC_1164.ALL;
4  entity main is
5      PORT(
6          -- custom ports go here (i.e. I/O)
7          btnL : in std_logic; -- button left
8          btnU : in std_logic; -- button up
9          led : out std_logic_vector(0 to 15); -- 16 leds
10         sw : in std_logic_vector(0 to 15); -- 16 switches
11         clk : in std_logic;
12     );
13 end main;
14 architecture behavior of main is
15     signal clk :std_logic := '0'; -- in
16     signal ps_ing_coffee :std_logic := '0'; -- in
17     signal t_grind_coffee :std_logic := '1'; -- in
18     signal p_ing_coffee :std_logic; -- out
19
20 begin
21     -- instantiation of entities
22     testbench : entity work.testbench port map(clk => clk
23         , ps_ing_coffee => ps_ing_coffee
24         , t_grind_coffee => t_grind_coffee
25         , p_ing_coffee => p_ing_coffee);
26     -- connection of entities by their ports
27     -- custom code here
28     t_start_transition <= '1' when (btnC = '1' and sw(14) =
           '0') else '0';
29     t_reqd_Milk <= '1' when (btnL = '1' and sw(14) = '0')
           else '0';
30     -- 3 further transitions are bound to a button
31
32     led(1)  <= (sw(1) and p_Milk_Heating) or (sw(0) and
           p_Stopped);
33     led(0)  <= (sw(1) and p_Preparing_milk_heating_out) or (
           sw(0) and p_Stopping);
34     -- 13 further places are bound to an LED + Switch
35 end;
```

Listing 2. VHDL skeleton code for place ing_coffee connected to transition grind_coffee

```
1  library IEEE;
2  use std.textio.all;
3  use IEEE.STD_LOGIC_1164.ALL;
4  entity testbench is
5      PORT(
6          clk : in std_logic := '0';
7          p_ing_coffee : out std_logic := '0';
8          ps_ing_coffee : in std_logic := '0';
9          t_grind_coffee : in std_logic := '1'     );
10 end testbench;
11 architecture behavior of testbench is
12     signal ing_coffeeir :std_logic;
13     signal ing_coffeeo2 :std_logic;
14 begin
15     ing_coffee : entity work.place generic map(1, 0) port
           map(ps_ing_coffee, ing_coffeeir, ing_coffeeo2, clk)
           ;
16     ing_coffeeir <= (ing_coffeeo2 and t_grind_coffee); --
           grind_coffee
17     p_ing_coffee <= ing_coffeeo2;
18 end;
```

Listing 3. VHDL (internal) implementation code for place ing_coffee connected to transition grind_coffee

Finally, in our test-setup we utilized Vivado-SDK-2016.2 to synthesize the bitstream for the Basys3 Artix-7 FPGA, which contains 33,280 logic cells in 5200 slices, with each slice containing four 6-input LUTs and 8 flip-flops. With this setup, the size-impact of the Petri net can be described as marginal, as can be seen in Figure 5.

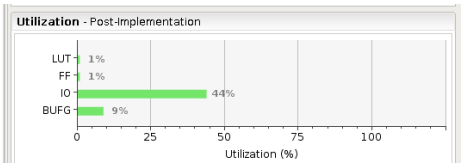


Figure 5. Resource utilization of the Petri net on a Basys 3 Artix-7 FPGA. The high I/O usage is due to our test-setup. The only required I/O is a clock.

## VI. Conclusion and Future Work

In this article, we showed how Adaptive Petri nets can be embedded in a workflow to synthesize Petri nets for context adaptive circuits. Adaptive Petri nets support a Petri net developer with a new tool, which helps to express intentions more directly and make context-awarenes a higher level language construct of Petri nets. We claim that directly expressing the adaptivity behavior of the net, allows developers to better collaborate and communicate with each other. By maintaining the ability to flatten these nets into standard Petri nets with inhibitor arcs, existing tools and model checking solutions can still be applied on this new class of nets. Because of the specific structure of APN, inhibitor arcs can be removed in most cases to extend the suitable tools and model checking capabilities even further.

Compared to the initial paper on APN, the concept was slightly improved to support commutative flattening of multiple APN configurations. Further, this article proposed a methodology of development for context adaptive FPGA-based applications. The algorithm to flatten Adaptive Petri nets to FPGA is extending the existing work of code generation from Petri nets for FPGA, not only by supporting a new class of nets, but also by supporting new kinds of composition operations and supporting the usage of statemachines as input.

The workflow, for synthesizing Petri nets to FPGA, is generic and allows an instantiation with several tools and techniques. We showed how a coffee machine model can be transformed. The coffee machine is context dependent on the user input and changes its behavior based on the selection. The transformation workflow utilizes model checking to verify the correctness through automated checks, manual checks, and to optimize the resulting circuit by eliminating dead places and transitions as well as invariants. It was shown that the resulting circuit is relatively small compared to the size of modern FPGA.

While the coffee machine was utilized here as an illustrative example, we already experimented with utilizing Adaptive Petri nets for human-aware robotic control [54], [55] by implementing the Haddadin automaton [56] as the controlling net for an Adaptive Petri net. In the future, Adaptive Petri nets should be directly synthesized on the FPGA, with partial dynamic reconfiguration, which most modern FPGA support. We are implementing further semantics for exception handling [54], which allows to set and reset the tokens inside a configuration point. Utilizing the similar runtime semantics of Adaptive Petri nets and role oriented programming languages to model and verify these languages [57].

## Acknowledgment

We gratefully acknowledge support from the German Excellence Initiative via the Cluster of Excellence "Center for advancing Electronics Dresden" (cfAED).

This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 692480. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Netherlands, Spain, Austria, Belgium, Slovakia."

## References

[1] C. Mai, R. Schöne, J. Mey, T. Kühn, and U. Aßmann, "Adaptive Petri nets – a Petri net extension for reconfigurable structures," in The Tenth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2018). IARIA XPS Press, 2018, pp. 15–23.

[2] J. Deepakumara, H. M. Heys, and R. Venkatesan, "FPGA implementation of MD5 hash algorithm," in Canadian Conference on Electrical and Computer Engineering 2001. Conference Proceedings (Cat. No. 01TH8555), vol. 2. IEEE, 2001, pp. 919–924.

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," ACM Computing Surveys (CSUR), vol. 22, no. 4, 1990, pp. 299–319.

[4] A. V. Yakovlev and A. M. Koelmans, "Petri nets and digital hardware design," in Lectures on Petri Nets II: Applications. Springer, 1998, pp. 154–236.

[5] N. Marranghello, "Digital systems synthesis from Petri net descriptions," DAIMI Report Series, vol. 27, no. 530, 1998.

[6] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 4, no. 2, 2009, p. 14.

[7] J. Padberg and L. Kahloul, "Overview of reconfigurable Petri nets," in Graph Transformation, Specifications, and Nets. Springer, 2018, pp. 201–222.

[8] R. Valk, "Object Petri nets," in Lectures on Concurrency and Petri Nets, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2003, pp. 819–848.

[9] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker," Electronic Notes in Theoretical Computer Science, vol. 71, 2004, pp. 162–187.

[10] J. Padberg and A. Schulz, "Model checking reconfigurable Petri nets with Maude," in Graph Transformation, ser. Lecture Notes in Computer Science. Springer, 2016, pp. 54–70.

[11] J. Padberg, "Reconfigurable Petri nets with transition priorities and inhibitor arcs," in Graph Transformation. Springer, 2015, pp. 104–120.

[12] M. Llorens and J. Oliver, "Structural and dynamic changes in concurrent systems: Reconfigurable Petri nets," IEEE Transactions on Computers, vol. 53, no. 9, 2004, pp. 1147–1158.

[13] J. Li, X. Dai, and Z. Meng, "Improved net rewriting systems-based rapid reconfiguration of Petri net logic controllers," in 31st Annual Conference of IEEE Industrial Electronics Society IECON., 2005, pp. 2284–2289.

[14] R. Valk, "Self-modifying nets, a natural extension of Petri nets," in Automata, Languages and Programming. Springer, 1978, pp. 464–476.

[15] S.-U. Guan and S.-S. Lim, "Modeling adaptable multimedia and self-modifying protocol execution," Future Generation Computer Systems, vol. 20, no. 1, 2004, pp. 123–143.

[16] A. Bukowiec and M. Doligalski, "Petri net dynamic partial reconfiguration in FPGA," in Computer Aided Systems Theory - EUROCAST, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2013, pp. 436–443.

[17] R. Muschevici, D. Clarke, and J. Proenca, "Feature Petri nets," in Proceedings 1st International Workshop on Formal Methods in Software Product Line Engineering (FMSPLE 2010), 2010.

[18] R. Muschevici, J. Proença, and D. Clarke, "Feature nets: Behavioural modelling of software product lines," Software & Systems Modeling, vol. 15, no. 4, 2016, pp. 1181–1206.

[19] E. Serral, J. De Smedt, M. Snoeck, and J. Vanthienen, "Context-adaptive Petri nets: Supporting adaptation for the execution context," Expert Systems with Applications, vol. 42, no. 23, 2015, pp. 9307 – 9317.

[20] H. Yang, C. Lin, and Q. Li, "Hybrid simulation of biochemical systems using hybrid adaptive Petri nets," in Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, pp. 42:1–42:10.

[21] L. Gomes and J. P. Barros, "Structuring and composability issues in Petri nets modeling," IEEE Transactions on Industrial Informatics, vol. 1, no. 2, 2005, pp. 112–123.

[22] S. S. Patil, "Coordination of asynchronous events," Ph.D. dissertation, Massachusetts Institute of Technology, 1970.

[23] C. A. Petri, "Kommunikation mit Automaten," Ph.D. dissertation, Universität Hamburg, 1962.

[24] T. Agerwala, "Special feature: Putting Petri nets to work," Computer, vol. 12, no. 12, 1979, pp. 85–94.

[25] K. Moore and S. Gupta, "Petri net models of flexible and automated manufacturing systems: a survey," International Journal of Production Research, vol. 34, no. 11, 1996, pp. 3001–3035.

[26] C. E. Cummings, "The fundamentals of efficient synthesizable finite state machine design using nc-verilog and buildgates," in Proceedings of International Cadence Usergroup Conference, 2002, pp. 1–27.

[27] S. Chevobbe, R. David, F. Blanc, T. Collette, and O. Sentieys, "Control unit for parallel embedded system." in ReCoSoC, 2006, pp. 168–176.

[28] N. Marranghello, "A dedicated reconfigurable architecture for implementing Petri nets," in M. Adamski (Ed.) Proceedings of the 2nd IFAC International Workshop on Discrete Event Systems Design, 2004, pp. 189–193.

[29] M. Adamski and M. Wegrzyn, "Petri nets mapping into reconfigurable logic controllers," Electronics and Telecommunications Quarterly, vol. 55, 2009, pp. 157–182.

[30] J. Carmona, J. Cortadella, V. Khomenko, and A. Yakovlev, "Synthesis of asynchronous hardware from Petri nets," in Lectures on Concurrency and Petri Nets. Springer, 2004, pp. 345–401.

[31] I. Grobelna, "Control interpreted Petri nets-model checking and synthesis," in Petri Nets - Manufacturing and Computer Science, P. Pawlewski, Ed. INTECH Open Access Publisher, 2012.

[32] T. Murata, "Petri nets: Properties, analysis and applications," Proceedings of the IEEE, vol. 77, no. 4, 1989, pp. 541–580.

[33] T. Kozlowski, E. Dagless, J. Saul, M. Adamski, and J. Szajna, "Parallel controller synthesis using Petri nets," in Computers and Digital Techniques, IEE Proceedings-, vol. 142. IET, 1995, pp. 263–271.

[34] E. Pastor and J. Cortadella, "Efficient encoding schemes for symbolic analysis of Petri nets," in Proceedings of the Conference on Design, Automation and Test in Europe, ser. DATE '98. IEEE Computer Society, 1998, pp. 790–795.

[35] S. Bulach, The design and realization of a custom Petri net based programmable discrete event controller. Aachen : Shaker, 2002.

[36] L. Gomes, "On conflict resolution in Petri nets models through model structuring and composition," in INDIN'05. 2005 3rd IEEE International Conference on Industrial Informatics, 2005. IEEE, 2005, pp. 489–494.

[37] R. Wiśniewski, G. Bazydło, L. Gomes, and A. Costa, "Dynamic partial reconfiguration of concurrent control systems implemented in FPGA devices," IEEE Transactions on Industrial Informatics, vol. 13, no. 4, 2017, pp. 1734–1741.

[38] L. Kahloul, S. Bourekkache, and K. Djouani, "Designing reconfigurable manufacturing systems using reconfigurable object Petri nets," International Journal of Computer Integrated Manufacturing, vol. 29, no. 8, 2016, pp. 889–906.

[39] D. Zaitsev and Z. Li, "On simulating turing machines with inhibitor Petri nets," IEEJ Transactions on Electrical and Electronic Engineering, 2017, pp. 147–156.

[40] N. Busi, "Analysis issues in Petri nets with inhibitor arcs," Theoretical Computer Science, vol. 275, no. 1, 2002-03-28, pp. 127–177.

[41] R. Lipton, "The reachability problem requires exponential space. department of computer science," Research Report 62, Yale University, Tech. Rep., 1976.

[42] K. Schmidt, "LoLA a low level analyser," in Application and Theory of Petri Nets, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2000, pp. 465–474.

[43] K. Wolf, "Petri net model checking with LoLA 2," in International Conference on Applications and Theory of Petri Nets and Concurrency. Springer, 2018, pp. 351–362.

[44] F. Kordon, H. Garavel, L. Hillah, E. Paviot-Adet, L. Jezequel, F. Hulin-Hubard, E. G. Amparore, M. Beccuti, B. Berthomieu, H. Evrard, P. G. Jensen, D. L. Botlan, T. Liebke, J. Meijer, J. Srba, Y. Thierry-Mieg, J. van de Pol, and K. Wolf, "MCC'2017 - the seventh model checking contest," Transactions on Petri Nets and Other Models of Concurrency (ToPNoC), vol. XIII, 2018, pp. 181–209.

[45] B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets," International Journal of Production Research, vol. 42, no. 14, 2004, pp. 2741–2756.

[46] J. P. Barros and L. Gomes, "Net model composition and modification by net operations: A pragmatic approach," in 2nd IEEE International Conference on Industrial Informatics, INDIN, 2004, pp. 309–314.

[47] M. Volkmann, "Integration von adaptiven Petrinetzen in ein Petrinetz Kompositions-system," Bachelor's thesis, Technische Universität Dresden, 2018.

[48] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in European Conference on Object-Oriented Programming. Springer, 1997, pp. 419–443.

[49] N. Cardozo, J. Vallejos, S. González, K. Mens, and T. D'Hondt, "Context Petri nets: Enabling consistent composition of context-dependent behavior." PNSE, vol. 12, 2012, pp. 156–170.

[50] L. Gomes and J. P. Barros, "Structuring and composability issues in Petri nets modeling," IEEE Transactions on Industrial Informatics, vol. 1, no. 2, 2005, pp. 112–123.

[51] N. Busi and G. M. Pinna, "Synthesis of nets with inhibitor arcs," in CONCUR'97: Concurrency Theory. Springer, 1997, pp. 151–165.

[52] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," in 2009 International Conference on Field Programmable Logic and Applications. IEEE, 2009, pp. 498–502.

[53] E. Soto and M. Pereira, "Implementing a Petri net specification in a FPGA using VHDL," in Design of embedded control systems. Springer, 2005, pp. 167–174.

[54] M. Jakob, "Extending adaptive Petri nets with a concept for exception handling," Master thesis, Technische Universität Dresden, 2019.

[55] H. Schole, "Modellierung von sensitivem roboterverhalten in szenarien der mensch-roboter-interaktion auf basis von kollaborationszonen," Master thesis, Technische Universität Dresden, 2019.

[56] S. Haddadin, M. Suppa, S. Fuchs, T. Bodenmüller, A. Albu-Schäffer, and G. Hirzinger, "Towards the robotic co-worker," in Robotics Research. Springer, 2011, pp. 261–282.

[57] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann, "A metamodel family for role-based modeling and programming languages," in International Conference on Software Language Engineering. Springer, 2014, pp. 141–160.