Adaptive Petri Nets – A Petri Net Extension for Reconfigurable Structures

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

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

***Abstract*—Petri nets are used to formally model the behavior of systems. However, when these systems dynamically change, e.g., due to context dependence, modeling gets complex and cumbersome since Petri nets are low-level and can not express dynamic changing parts. Expressing dynamically changing parts of the system directly within Petri nets increases the clarity and allows for modeling complex, context dependent, systems. While various approaches can be found in the literature, their integration into the Petri net ecosystem is often not considered. This restricts the available tools and analysis techniques to those, which can handle that custom net type. We present adaptive Petri nets, an extension to Petri nets, which directly expresses variability within the net. Our approach integrates well with other Petri net extensions, such as colored tokens, inhibitor arcs or hierarchy. Most importantly, it is possible to convert an adaptive Petri net to a semantically equivalent Petri net with inhibitor arcs. This work presents the formalism of adaptive Petri nets, how they can be flattened to Petri nets with inhibitor arcs and their graphical representation. The feasability and usability is demonstrated on two examples that are modeled, flattened and analyzed.**

Keywords–Petri nets; Reconfigurable Petri nets; Inhibitor Arcs; Analysis

I. Introduction

Petri nets are a mathematical modeling technique used in many areas. Their strengths are especially in modeling concurrent, asynchronous, distributed, parallel, or nondeterministic systems [1]. Based on a mathematical model, they can be analyzed for various properties, such as deadlocks, reachability, or boundedness [2]. While the graphical notation of Petri nets reduces the learning curve and improves communication in teams. In general, Petri nets tend to get large, making it difficult to work on them. Various syntactic additions exist to improve readability and their expressiveness, while still allowing to flatten the net into a semantically equivalent Petri net without these additions. Examples are hierarchical structuring [3], composition [4], or colored tokens [5].

Our approach, adaptive Petri nets, is a Petri net extension allowing a net designer to model structures, which change at runtime. These nets are reconfigured by configuration places that enable or disable parts of the net. Consequently, the net designer can express his/her intentions directly. Additionally, it might open the door for analysis techniques, which utilize the added semantic information.

The paper is structured as follows. In Section II, the related work is reviewed. Next, in Section III-A the formal models of Petri nets and Petri nets with inhibitor arcs are introduced. Two examples from the literature motivate the need for adaptive Petri nets in Section IV. Section V explains the concept of adaptive Petri nets together with a formal semantic and graphical syntax. An algorithm for flattening will show how adaptive nets can be reduced to Petri nets with inhibitor arcs. After that, the two examples from Section IV are reimplemented with our notation. Here, we show that the Petri net analyzers LoLa [2] and Tina [6] can analyze the flattened version of adaptive nets.

II. Related Work

Reconfigurable Petri nets can be seen as composition at runtime. In [4], this is called dynamic composition and is characterized as "*rare*", because it "*radically changes the Petri net semantics* and *complicates the available analysis techniques*". Regardless, several approaches to implement *dynamic composition* exist.

Object Petri nets [7] 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 [8] 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** [9], [10] 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 can not 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) [11]. 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 amount of places and transitions unchanged, i.e., only the flow relation can be changed. Flattening increases the

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

size of transitions significantly, i.e., by the amount of transitions multiplied by the number of reconfigurations. With **improved net rewriting systems** [12], 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 [13] were already introduced in 1978 to permit reconfiguration at runtime. Arcs between places and transitions are annotated with a weight specifying the amount of tokens required inside the place until the transition becomes enabled. To achive reconfiguration, these weights are made dynamic by linking them to a place. The number of the weight is then determined by the amount 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 [13].

Guan et al. [14] proposed a dynamic Petri net, which creates new structures when firing transitions. The net is divided in a control and a presentation net. The control net changes the structure of the presentation net by annotations on its nodes. Verification and reducibility were explicitly excluded by the authors.

A practical example was shown in **Bukowiec et al.** [15], who modeled a dynamic Petri net, which could exchange parts of the net 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 Programmabe Gate Array (FPGA). Since this work was of more practical nature, the reconfiguration and transformation was not formalized. Although, it was shown by Padberg et al. [9] that this kind of net can be transformed into a representation, which can be verified using Maude.

Dynamic Feature Petri nets (DFPN) [16] 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 [17]. Compared to adaptive Petri nets, this type of net is problem specific and has the limitation of indirection by boolean formulas. A boolean formula can not 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 amount of tokens inside a place.

With **Context-adaptive Petri nets** [18], 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 was not detailed how the analysis is implemented, therefore scalability is unclear. Additionally, the flattening of these nets is not supported.

Hybrid Adaptive Petri Nets [19] 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.

We found that most of the existing work lacks a good integration in the Petri net ecosystem. The reconfiguration is either written as graph rewrite rules or external descriptions, which fit Petri nets more from a theoretical point of view but not for modelling. Flattening these nets to a lower level Petri net is often not the goal of the approaches, hence existing Petri net tools can not be used, e.g., for efficient model checking or code generation.

III. PRELIMINARIES

In this section, definitions and notations are introduced, which are used throughout the paper.

A. Petri Net Definitions

This section recalls the definition of Petri nets and establishes the notation.

Definition 1: A **Petri net** [1] 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 a tuple element is done in dot notation: for a Petri net Σ, 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 $x\bullet = \{y | (x, y) \in F\}$.

For example, $t\bullet$ 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 $M : P \to \mathbb{N}$.

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 an initial marking, every reachable marking $M(p) \leq k$, we speak of p as **k-bounded**. This place never contains more than k tokens. If k equals 1, this place is called **safe**.

B. Inhibitor Arcs

Inhibitor arcs extend the flow relation in Petri nets by an arc, which will disable a transition when the connected place has a specified amount of tokens in it. A Petri net with inhibitor arcs is more expressive than a normal Petri net. For example, a Petri net with inhibitor arcs can implement a Turing machine [20], while this is not possible with standard Petri nets. This affects the available tools for model checking, for example, the halting problem can not be solved in general for Turing-complete languages.

Definition 7: The set of **inhibitor arcs** $I \subseteq (P \times T)$ is added to Def. 1. An **Inhibitor Petri net** is a tuple

$\Sigma = (P, T, F, I, W, M_0)$. $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$. $F \subseteq (P \times T) \cup (T \times P)$, $I \subseteq (P \times T)$. $W : (F \cup I) \to \mathbb{N}$, $M_0 : P \to \mathbb{N}$.

Definition 8: The weight W is extended to also include the inhibitor arcs $W : (F \cup I) \to \mathbb{N}$.

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

Definition 9: A transition t is **enabled**$_i$, iff all places connected by an inhibitor arc are below the weight $M(p) < W(p, t)$ for all $p \in \circ t$ and the transition is enabled as defined in Def. 4.

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: —○.

IV. Motivating Examples

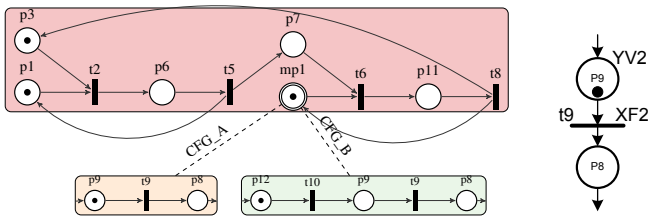
This section will show examples from the literature to both motivate the need for reconfiguration inside Petri nets and use them to demonstrate adaptive Petri nets in Section VI. The first example shows an informal reconfiguration model of a controller [15] and the second example is a coffee machine implemented with Dynamic Feature Nets [16].

A. Dynamic Control Structures

Control structures for circuits are typically modeled with finite state machines. However, if parallelism or asynchronism is needed, Petri nets are employed [21], [22]. The modeled Petri net gets converted into a hardware description language to load it onto an FPGA. To support modern FPGA with partial dynamic reconfiguration, in which parts of the FPGA can be reconfigured at runtime, the Petri net should support reconfiguration at runtime, too. This is not directly possible with standard Petri nets but requires a reconfigurable addition.

In [15], a proposal was made to model the reconfiguration by two subnets (pages), which are exchanged based on an incoming signal. Their use case is a cement mixing machine, which can be configured with and without a water heating element. The type of Petri net they use is called control interpreted Petri net. These nets are specifically made for use in electronic circuits, so that they can send and receive signals, modeled in terms of variables. Each transition is annotated with a variable, which inhibits the firing until its value becomes true. A place can be annotated by a name, representing a variable, which will be set to true when the place contains a token. The net is compiled into a hardware description language that can be used to synthesize the circuit on the FPGA. An example for control interpreted Petri nets is shown in Figure 1b: the place *P9* enables the output signal *YV2*, if it contains a token, the transition *t9* fires only, if the input signal *XF2* is active and a token is inside *P9*.

We depict the example from the paper of Bukowiec et al. [15] here to show how their reconfigurable Petri nets are implemented. In their work, a cement mixing machine was modeled with a Petri net. Each transition will trigger valves or motors to support the cement mixing. The exact working is irrelevant here, except that the cement can be mixed with either heated or cold water. These two features are mutually exclusive and should replace the corresponding logic on the FPGA by partial dynamic reconfiguration. A simplified version of the Petri net for the cement mixing machine can be seen in Figure 1a. As simplification the annotations and non-branching structures were removed. This was only done for readability. The pages of Figure 1a are chosen by configuration signals *CFG_A* and *CFG_B*. *CFG_A* enables the default behavior of the water element, while *CFG_B* is the behavior of the water heater. The configuration signals can be found in Figure 1a, while in the original paper, they were part of the informal description. The *CFG*-signals are either sent by an algorithm of the controler or comes from the manual input of a human supervisor.



(a) Simplified net of Bukowiec (without annotations, added CFG_* arcs)

(b) Control Interpreted Petri net

Figure 1. Dynamic control structures from [15]

The resulting net was synthesized to an FPGA specification with a reserved reconfigurable area, in which the pages corresponding to *CFG_A* and *CFG_B* are synthesized.

B. Dynamic Features

Product line engineering is an important topic in software development. A software product line (SPL) is a collection of software systems with a shared set of assets. Typically, there exists a core component in a product line, which gets enriched by various features. These features can be either applied at compile time (static) or at runtime (dynamic). Modeling an SPL with a Petri net requires representing the core and its features, so that they can be enabled or disabled based on the configuration.

In [23], (dynamic) Feature Oriented Petri nets (FOP) were proposed to model an SPL with Petri nets. The activation of a feature is encoded as a boolean variable. The nodes and arcs in the Petri net are annotated by logical formulas containing the feature variables. If the formula evaluates to *false*, the node or arc is temporarily removed from the net until the formula evaluates to *true* again. For static features the variable assignment comes from the outside and does not change while the Petri net gets executed. Therefore, all formulas with static features can be evaluated at first. To model dynamic features, a transition can be annotated by assignments to feature variables. For that, the annotation is split into a formula and an assignment part, illustrated in Listing 1. This transition fires only if the feature *Milk* is deactivated and *Coffee* activated. When the transition fires, it enables the feature *Milk*. This transition is then temporarily removed from the net, as its formula no longer evaluates to true.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

Listing 1 Example Formula Annotation

```
1: ¬Milk∧Coffee
     Milk On
```

The running example for [23] is a configurable coffee machine, showcased in Figure 2. This net models a coffee machine, which can get a milk module added at runtime. Adding the milk module is done as a dynamic feature inside the net and triggered by the *connect* and *disconnect* transitions.

V. Concept of Adaptive Petri Nets

With adaptive Petri nets, we propose a concept, which supports a Petri net developer to enable and disable a subset of nodes based on the amount of tokens in a set of places. Ultimately, our goal is to support the development of Petri nets with dynamic changing behavior while still supporting the flattening to inhibitor Petri nets to allow the use of standard Petri net tools.

An adaptive Petri net extends the Petri net definition by a set of configuration points $C = \{c_1, c_2, \ldots\}$. A configuration point will enable or disable parts of a Petri net Σ.

Definition 10: An **adaptive Petri net** 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 11: A **configuration point** is a tuple $c = (p, w, N)$ referencing the nodes of a containing Petri net Σ.

- $p \in \Sigma.P$, 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

Definition 12: The set of **external nodes** ($E \subseteq N$) are nodes of N which are connected to nodes outside of N. $E = \{x | x \in N \wedge (\exists y \in ((P \cup T) \setminus N)(\{(x, y), (y, x)\} \cap F \neq \emptyset)\}$

Definition 13: The set of **internal nodes** for a configuration point is calculated by $I = N \setminus E$.

An example for an adaptive Petri net can be seen in Figure 3. The configuration points are $c_1 = (pc1, 1, \{p1, t1, p2, t3\})$ and $c_2 = (pc2, 1, \{p1, t2, p3, t4\})$. The set of externel nodes for c_1 is $c_1.E = \{p_1, t_3\}$, while the set of internal nodes is $c_1.I = \{t_1, p_2\}$.

Definition 14: A configuration point $c \in C$ is **enabled**, iff $(c.w > 0 \wedge M(c.p) \geq c.w) \vee (c.w < 0 \wedge M(c.p) < |c.w|)$. With *M* being the marking function of Def. 3. As a shorthand we will refer to the set of enabled configuration points as $C_e \subseteq C$.

An enabled adaptive Petri net will not change the behavior of the net, while a disabled adaptive Petri net stops the flow of tokens from E to N. This changes the firing definition of Def. 5 and the enabling definition of Def. 4. This is defined in Defs. 17 and 18.

We want to navigate from a place or transition to all configuration points, which are containing this node. For this we define the following functions.

Definition 15:
- The set of configuration points a node belongs to is defined by the function $B^N : (P \cup T) \to \mathbb{P}(C)$ 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: $B^E : (P \cup T) \to \mathbb{P}(C)$ 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: $B^I : (P \cup T) \to \mathbb{P}(C)$ with $B^I(n) = \{c | c \in C \wedge n \in c.I\}$.

Definition 16: 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 ${}_c\bullet t = {}\bullet t \setminus \{p | c \in (B^E(t) \setminus C_e) \wedge p \in c.E\}$, respectively.

Definition 17: 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 $p \in {}_c\bullet t$ is decremented by $W(p, t)$. The fire semantics of all other transitions follows Def. 5.

Definition 18: A transition $t \in T$ is **enabled**$_a$, iff it is enabled according to Def. 4 and the following condition holds true $\{p | p \in {}\bullet t \wedge p \in c.E;\ \forall c \in (B^I(t) \setminus C_e)\} = \emptyset$.

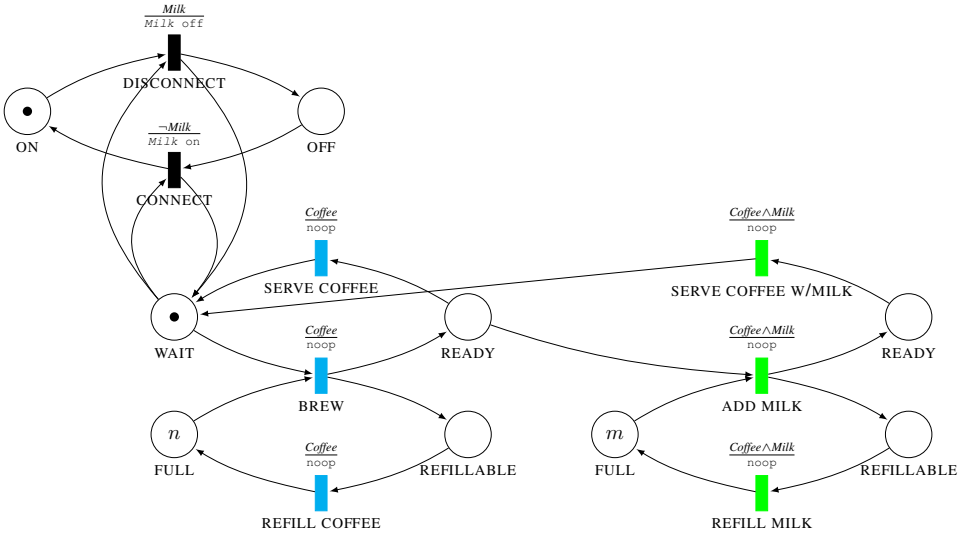


Figure 2. Dynamic Feature Petri net from [16]

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.I 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)
11:                 F ← F \ ((t_2 × c.N) ∪ (c.E × t_2))
12:                 ConnectByArc((⊤, c, t, F, I, W))
13:                 ConnectByArc((⊥, c, t_2, F, I, W))
14:             end if
15:         end for
16:         C ← C \ {c}
17:     end for
18: end procedure
```

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

```
1:  procedure CONNECTBYARC((e, c, t, F, I, W))
2:      if (c.w > 0 ∧ e = ⊤) ∨ (c.w < 0 ∧ e = ⊥) then
3:          F ← F ∪ {(c.p, t), (t, c.p)}
4:          W(c.p, t) ← |c.w|
5:          W(t, c.p) ← |c.w|
6:      else
7:          I ← I ∪ {(c.p, t)}
8:          W(c.p, t) ← |c.w|
9:      end if
10: end procedure
```

In Def. 18, we prohibit that new tokens enter from E to N. For the case, when the external node is a place ($p \in E$) and targets an internal transition ($t \in I$), by inhibiting the transition. For all other cases, Def. 17 changes the places from which tokens are removed and where tokens are added after firing. A transition, which belongs to a disabled configuration point, can not remove tokens from any place in E of this configuration point ($p \in c.E$) and can not create any tokens in any place of N of this configuration point ($p \in c.N$).

A. Flattening Algorithm

Special attention was given to the ability to remove the configuration point structure and replace it with Petri net structures of lower level Petri nets to be compatible with existing Petri net tools. This feature reduction is also called flattening and was already shown for different concepts. Colored Petri nets were introduced by Jensen [5] and are reducible to standard Petri nets by net duplication. Huber [3] published a paper enhancing colored Petri nets with hierarchy and showed how they can be transformed to standard Petri nets with flattening. Portinale [24] describes an or-transition, which, contrary to the standard transition, contains or-logic instead of and-logic. This transition can be reduced to Petri nets with inhibitor arcs.

Theorem 1: An adaptive Petri net can be flattened to a semantically equivalent Petri net with inhibitor arcs $\Sigma = (P, T, F, W, M_0, I)$.

Algorithm 3 Helper method to duplicate a transition

```
1:  procedure DUPLICATE((t, P, T, F, W, C, 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:      for ∀c ∈ C do
9:          if t ∈ c.N then
10:             c.N ← c.N ∪ {t_2}
11:         end if
12:     end for
13: end procedure
```

The flattening of Theorem 1 is described in Algorithm 1. We have to show that flattening will respect Defs. 17 and 18.

Lemma 1: ConnectByArc of Algorithm 2 with $e = \top$ will disable t when the configuration point c is disabled.

Proof: We show the correctness of Lemma 1 for the if-branch with $c.w > 0$ in lines 3-5 of Algorithm 2 and the else-branch with $c.w < 0$ in lines 7-8. The if-branch will disable t when $M(c.p) < c.w$, which is the same condition when c is disabled according to Def. 14 ($M(c.p) \geq c.w$). The else-branch will disable t when $M(c.p) \geq |c.w|$, which is the same condition when c is disabled according to Def. 14 ($M(c.p) < |c.w|$) □.

Lemma 2: ConnectByArc of Algorithm 2 with $e = \bot$ will disable t when the configuration point c is enabled.

Proof: The correctness of Lemma 2 is analogous to Lemma 1. It has to be shown for the if-branch with $c.w < 0$ in lines 3-5 of Algorithm 2 and the else-branch with $c.w > 0$ in lines 7-8. The if-branch will disable t when $M(c.p) \geq |c.w|$, which is the same condition when c is disabled according to Def. 14 ($M(c.p) < |c.w|$). The else-branch will disable t when $M(c.p) < c.w$, which is the same condition when c is disabled according to Def. 14 ($M(c.p) \geq c.w$) □.

Lemma 3: A disabled configuration point will disable the firing of all internal transitions, which are in the postset of an external place. According to Def. 18.

Proof: The set of places and transitions, which are referred by Def. 18, are selected in lines 3-4 of Algorithm 1 ($\forall p \in c.E \cap P$ and $\forall t \in p\bullet \cap c.I$). On Line 5, *ConnectByArc* will disable the transition when c is disabled as shown with Lemma 1□.

Lemma 4: The algorithm will transform all transitions, which have a configured postset or preset as defined in Def. 16 and utilized in Def. 18, i.e., $t\bullet \neq t\bullet_c \vee \bullet t \neq \bullet_c t$.

Proof: Def. 17 only changes external transitions, which is implemented in Line 8 of Algorithm 1. Only those transitions have to be changed, which have $t\bullet_c \neq t\bullet$ or $\bullet_c t \neq \bullet t$. This is implemented in Line 9 with a logical *or*. The left part of the *or* is $t\bullet \cap c.N \neq \emptyset$, which is equivalent to the definition of $t\bullet_c$. The right part of the *or* is $\bullet t \cap c.E \neq \emptyset$, which is equivalent to the definition of $\bullet_c t$□.

Lemma 5: A flattened transition will only produce tokens in $t\bullet_c$. According to Def. 17.

Lemma 6: A flattened transition will only consume tokens from $\bullet_c t$. According to Def. 17.

Proof: As shown in Lemma 4, the set of transitions is selected correctly. We will now proof that the tokens are produced and consumed according to Def. 17.

The code in Line 10 of Algorithm 1 duplicates the transition according to Algorithm 3. This will take the original transition t and create a new transition t_2 with the same properties, connected arcs and inhibitor arcs together with their weights. Additionally, t_2 is added to all N in which t was contained.

The flow relation of t_2 is updated in Line 11, to remove all $c.E$ from its preset and all $c.N$ from its postset, as defined in Def. 16. On Line 12 the transition t without the altered flow relation will fire only when the configuration point is enabled (see Lemma 1). Transition t_2 with the altered flow relation will only fire when the configuration point is disabled as defined on Line 13 (see Lemma 2)□.

Proof: By proving *Lemmas* $3, 5$ *and* 6, we could show that the semantics of adaptive Petri nets (Defs. 17 and 18) is preserved in a Petri net with inhibitor arcs □.

This shows that we can flatten arbitrary adaptive Petri nets into Petri nets with inhibitor arcs. The flattened net will duplicate transitions and add new arcs and inhibitor arcs to the net.

Lemma 7: When all $c \in C$ fulfill the condition $(c.w > 0) \wedge (c.E \cap T = \emptyset)$, an adaptive Petri net can be flattened without adding inhibitor arcs.

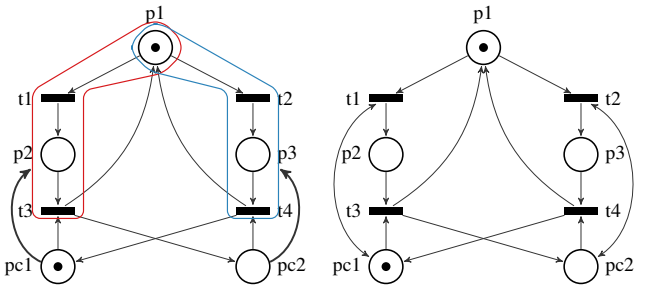
Proof: According to the condition $c.E \cap T = \emptyset$, the changes to the Petri net happen only on Line 5 of Algorithm 1. With the condition $c.w > 0$ only lines 4-5 of Algorithm 2 are executed, adding an incoming and outgoing arc to a transition□.

The expressive power of adaptive Petri nets is generally higher than that of Petri nets, since we flatten it to a Petri net with inhibitor arcs. A higher expressive power will make some properties unsolvable by model checkers. There are two methods to obtain a Petri net without inhibitor arcs from an adaptive Petri net. *Avoiding inhibitor arcs* at all, by fulfilling Lemma 7 or designing a net with only *bounded configuration places*. It was shown in [25] that an inhibitor arc can be replaced by an equivalent structure, if the inhibiting places are bounded. From Algorithm 1, we can see the only inhibiting places generated are the configuration places. Therefore, a Petri net designer can chose those places carefully or add additional structures to make sure these places are bounded.

B. Graphical Notation

To integrate configuration structures well within Petri nets, we can model a Petri net and then define all configuration points. This approach requires the net designer to manually update the configuration points each time a node was added or removed. A better approach is to create a graphical language, which integrates in the existing graphical language of Petri nets.

The graphical language must express each element of the tuple $c = (p, e, N)$. With $N \subseteq (P \cup T)$, we can draw a contour around all connected nodes of a configuration point forming an area. Since it is not required to have all nodes in N connected with each other, this would create multiple areas belonging to one configuration point. For that the areas for each configuration point should get a unique color or a unique



(a) Adaptive Petri net configured by $C = \{c_1, c_2\}$ with configuration points $c_1 = (pc1, 1, \{p1, t1, p2, t3\})$, $c_2 = (pc2, 1, \{p1, t2, p3, t4\})$.

(b) Adaptive Petri net flattened with Algorithm 1. According to Lemma 7, no inhibitor arcs were added.

Figure 3. Flattening of a simple adaptive Petri net

annotation. To declare the configuration point p together with the weight e, we will draw a bold arc or inhibitor arc from p to all areas of N. This arc can be annotated with a weight, which will become e. When an inhibitor arc is used, the weight must be multiplied with -1 to receive e.

An example for a simple Petri net with two configuration points can be seen in Figure 3. On the left is the adaptive Petri net with two different colored areas representing the nodes N of the configuration points c_1 and c_2. The net will execute alternatingly the net of c_1 and c_2, since the configuration points $pc1$ and $pc2$ are alternating their tokens. When flattening this net, the external nodes are $c_1.E = \{t1\}$ and $c_2.E = \{t2\}$.

VI. Use Cases Revisited

In this section, adaptive Petri nets are put to work. We will show the implementation of two examples from Section IV. It is shown how these examples can be represented in adaptive Petri net syntax. In the end, we will show how the flattening affects the size and model checking results.

A. Dynamic Control Structures

The converted Petri net of Figure 1a can be seen in Figure 4. The reimplementation in Figure 4 required some changes. Removing the initial token from *p9* and *p12* is necessary, because adaptive Petri nets would evaluate this token. Instead of the tokens, a place and transition (px, tx) were added before *p9* and *p12*. Another change was required for enabling transition *t6*. It should be enabled when either *p8* or *p82* contain a token. To achieve this, the transitions *ty1*, *ty2* and the place *py* were added. We argue, these changes could be automated if a formalization for the hierarchy concept of dynamic control structures was found, which can be flattened to a net with exactly this structure.

With adaptive Petri nets, the reconfiguration can be expressed inside the net. For the configuration variables *CFG_A* and *CFG_B*, two transitions were added, which are annotated according to *control interpreted Petri net* syntax. They can only fire when the signals *CFG_A* or *CFG_B* are active. Both transitions are connected to a single place *Conf*, which configures both configuration points - enabling one and disabling the other.

After modeling the original net with adaptive Petri nets, it is still possible to generate the hardware description language from it, as the annotations of control interpreted Petri nets are not prohibited in our model or modified in its semantics. Besides

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

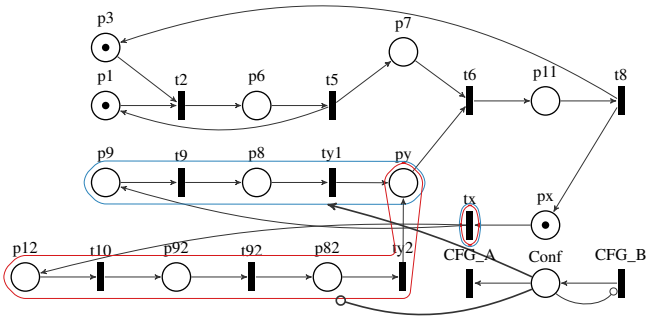


Figure 4. Dynamic control structures of [15] in adaptive Petri net notation. Configured by $C = \{c_1, c_2\}$ with $c_1 = (Conf, 1, \{p9, p8, t9\}, tx, ty1, py)$ and $c_2 = (Conf, -1, \{p12, t10, p92, t92, p82, tx, ty2, py\})$

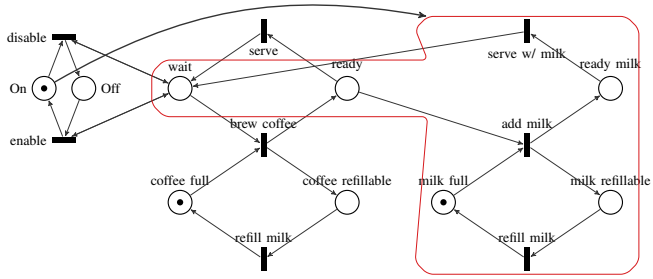


Figure 5. Dynamic Feature Petri net in adaptive Petri net notation Configured by $C = \{(On, 1, \{serve\ w/\ milk, ready\ milk, add\ milk, milk\ full, refill\ milk, milk\ refillable, wait, ready\})\}$.

this, through the net flattening of the configuration points c_1 and c_2, also older FPGA without dynamic reconfiguration can be targeted. Contrary to the original version, where the information regarding the reconfiguration signals was informally inside the description, the runtime semantics of this net is contained in the model. This example shows that adaptive Petri nets can be combined with other formal models of Petri nets (in this case *control interpreted nets*) as they are not imposing any restrictions.

B. Dynamic Features

We will show here how dynamic feature Petri nets [16] from Section IV can be modeled with adaptive Petri nets.

The example from Section IV can be converted straight forward to adaptive Petri nets. For each boolean variable inside a feature annotation we create one *configuration point* with this boolean variable as *configuration place*. All nodes, which are annotated with this boolean variable, are then put into the set N of this configuration point. In our example, the boolean variable *Milk* is mapped to the configuration place *On*. Then all nodes, which contain *Milk* in their annotated formula, are added to N, as well as the nodes bofore and after.

As can be seen in Figure 5, the resulting net has the same size and structure as the original. The annotations were exchanged for the graphical representation of the *configuration point*. Something that can not be expressed in feature nets are configurations based on integers. For example, the places *coffee full* and *milk full* basically are context information to decide whether a net is enabled or disabled. With adaptive nets, such a counting state can be utilized as *configuration place* to disable the net when the amount of token reaches zero.

TABLE I. SIZES OF NETS SHOWN IN SECTION VI

Net	Places	Transitions	Arcs
Original (Figure 4)	13	12	29
Flattened	13	13	32
Flattened (Inhibitor)	14	13	34
Original (Figure 5)	9	8	24
Flattened	9	8	26
Flattened (Inhibitor)	9	8	26

This example had shown that adaptive Petri nets are a feasible alternative for dynamic feature oriented Petri nets. This is especially helpful for cases where adaptivity is required, but no feature orientated development used. Adaptive nets have an advantage over feature Petri nets, which arbitrary places can configure the net, while in dynamic feature oriented Petri nets only formulas over boolean values are possible. Therefore, a layer of indirection is removed. In a recent paper [17], the authors showed how a feature net can be converted to a Petri net, including the formulas. Using this technique, feature nets can also be converted into an equal adaptive net.

C. Flattening and Analysis

One of the goals for adaptive Petri nets is to use existing Petri net tools for model checking. This goal can be reached by flattening all configuration points to Petri nets with inhibitor arcs. Since not all tools support inhibitor arcs, we will also flatten all inhibitor arcs with the algorithm of [25]. The only condition is that the place where the inhibitor arc originates must be bound. This condition holds true for all of our examples.

We flattened the Petri nets of our examples and used LoLa [2] as well as Tina [6] to analyze for common Petri net properties.The tool for flattening and analysis can be found online [26].

The results comparing the size can be seen in Table I. They should give an idea how the size of the nets will increase using flattening. As the size increase depends largely on the net structure, amount of configuration points and which nodes are marked as external, there is no general rule for the size increase. A configuration point consisting only of external places will not increase as much as a configuration point with external transitions. This is due to the fact that external transitions need to be duplicated. A bigger cost is the flattening of inhibitor arcs, which is especially big if the bound is larger than 1 as can be seen in the last row. Flattening an adaptive net will not always yield an inhibitor as can be seen in the row for Figure 5.

In Table II, the results of the model checking tools LoLa and Tina are shown. A short description on the checked properties: reversible (from every state of the net, we can reach the initial marking), deadlock free (starting from M_0, the net can always fire), k-bounded (all places contain at most k tokens) and live (all transitions and places can be reached from the initial marking). The column *markings* corresponds to all unique combinations of markings this net can reach. It is an indicator how long model checking will take, as for some properties (e.g., reversibility), all markings must be calculated.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

TABLE II. MODEL CHECKING RESULTS OF NETS SHOWN IN SECTION VI. [1] = TINA, [2] = LOLA

Net	reversible[12]	deadlock[12]	live[1]	k-bounded[1]	markings[12]
Figure 4	yes	no	yes	k=1	44
Figure 5	yes	no	yes	k=1	20

D. Other semantics

There are various alternative approaches to define a semantics for adaptive nets. We settled with a semantics, which requires only few changes to the original net, so that a more complex semantics might build on top of this.

Our semantics orients itself on most imperative programming languages. The exchange of a method (e.g., by pointer in C or by *invoke dynamic* in Java) will happen in a similar fashion. Only new calls to this method are influenced, while current running threads inside this method will still finish. When the program tries to call the method another time it will call its replacement.

Another disabling semantics we considered is to completely stop all token movement within the configured part. This can be implemented as an extension of Def. 18. When flattening this modified version, all transitions inside N have to be connected to the *configuration place*. A use case for this might be freezing an algorithm, e.g., in a single threaded environment, which switches to another thread.

Further extending this semantics, one might reset all tokens of N to an initial state. A similar approach was presented in [27] to implement exceptions in Petri nets.

VII. CONCLUSION

This paper presented a new Petri net extension for modeling dynamic parts inside a Petri net. Contrary to existing proposals this extension puts the least restrictions on the Petri net model. We do neither restrict to a composition model nor the specification language. It was shown that adaptive Petri nets can be specified formally and graphically. The biggest advantage of adaptive Petri nets is the possibility of flattening an adaptive net to a Petri net with inhibitor arcs. By this, existing Petri net tools can be reused for this model, e.g., for code generation or model checking. Because of the specific structure of adaptive nets, inhibitor arcs can be removed in most cases. This was shown on two examples, which were analyzed by low level Petri net tools.

In our ongoing work, adaptive Petri nets are used to convert the control flow of a role-oriented programming language (SCROLL [28]) to Petri nets, as well as generating control structures for hardware / software codesign. Future work will integrate adaptive Petri nets with Petri net composition models and improve tool support.

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

[2] 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.

[3] P. Huber, K. Jensen, and R. M. Shapiro, "Hierarchies in coloured Petri nets," in Advances in Petri Nets 1990, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1989, pp. 313–341.

[4] 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.

[5] K. Jensen, "Coloured Petri nets and the invariant-method," Theoretical Computer Science, vol. 14, no. 3, 1981, pp. 317–336.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1

[23] R. Muschevici, J. Proença, and D. Clarke, “Modular modelling of software product lines with feature nets,” in Software Engineering and Formal Methods. Springer, 2011, pp. 318–333.

[24] L. Portinale, “Behavioral Petri nets: a model for diagnostic knowledge representation and reasoning,” IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 27, no. 2, 1997, pp. 184–195.

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

[26] C. Mai. An implementation of Adaptive Petri nets. [Online]. Available: https://github.com/balrok/adaptive_pn (10.01.2018)

[27] H. Leroux, D. Andreu, and K. Godary-Dejean, “Handling exceptions in Petri net-based digital architecture: From formalism to implementation on FPGAs,” IEEE Transactions on Industrial Informatics, vol. 11, no. 4, 2015, pp. 897–906.

[28] M. Leuthäuser and U. Aßmann, “Enabling view-based programming with SCROLL: Using roles and dynamic dispatch for etablishing view-based programming,” in Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering, ser. MORSE/VAO ’15. ACM, 2015, pp. 25–33.

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-610-1