Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Master Thesis

Jan-Uriel Lorbeer

Hierarchical Reconfigurable Petri Nets

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jan-Uriel Lorbeer

Hierarchical Reconfigurable Petri Nets

Master Thesis eingereicht im Rahmen der Master Thesis

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Julia Padberg

Eingereicht am: 28.05.2018

Title of the paper
Hierarchical Reconfigurable Petri Nets

Keywords
Reconfigurable Petri nets, Petri net transformations, hierarchical Petri nets

Abstract
The challenging development of modern systems can be eased through the usage of appropriate models to simulate, evaluate and validate the system before hand. One well known method for this is the deployment of Petri nets. Especially challenging is the development of large systems with dynamic components. Hierarchical Petri nets (HPN) provide a more abstract view whereas reconfigurable Petri nets (RPN) allow dynamic structural adaptation. This thesis presents the combination of RPN and HPN yielding a hierarchical structure for reconfigurable Petri nets. The aim of this thesis is to: present a concept for a hierarchical reconfigurable Petri net (HRPN) model based on substitution transitions (with mathematical definition), prove the models correctness, implement HRPN within the tool ReConNet and evaluate the HRPN model and expanded ReConNet tool using a use case of a flexible manufacturing system.

Thema der Arbeit
Hierarchical Reconfigurable Petri Nets

Stichworte
Rekonfigurierbare Petri-Netze, Petri-Netz Transformationen, hierarchische Petri-Netze

Kurzzusammenfassung
Die anspruchsvolle Entwicklung moderner Systeme kann durch die Verwendung entsprechender Modelle zur Simulation, Evaluation und Validierung erleichtert werden. Eine bekannte Methode dafür ist der Einsatz von Petri-Netzen. Besonders anspruchsvoll ist die Entwicklung von großen Systemen mit dynamischen Komponenten. Hierarchische Petri-Netze (HPN) bieten eine abstraktere Sicht und Rekonfigurierbare Petri-Netze (RPN) erlauben dynamische strukturelle Adaption. Diese Thesis präsentiert die Kombination von RPN und HPN zu einer hierarchischen Struktur für rekonfigurierbare Petri-Netze. Das Ziel dieser Thesis ist: die Vorstellung eines Konzeptes für ein hierarchisches rekonfigurierbares Petri-Netz Model das auf Substitutions-Transitionen basiert (inklusive mathematischer Definition), das Beweisen der Korrektheit des Models, die Implementierung von HRPN in dem Werkzeug ReConNet und die Evaluation des HRPN Models und des erweiterten ReConNet Werkzeuges unter der Zuhilfenahme eines Anwendungsfalles eines flexiblen Manufaktursystems.

Contents

1. Introduction

The development of modern systems comes with many challenges. Some of these challenges can be coped with through the usage of appropriate models to simulate, evaluate and validate the system before hand. One well known and widely accepted method for this is the deployment of Petri nets [Pet81]. Petri nets provide a graphical language for constructing system models as well as a precise, mathematical semantics. With a well build model of the system faults in the system can be detected and fixed at earlier development stages. The earlier faults can be detected the easier and cheaper it is to fix them.

With a steady increase in complexity and size of modern systems their models also become larger and less comprehensible. A way to counteract this issue is the employment of a layer of abstraction such as hierarchy. Hierarchical Petri nets (HPN) [JR12] combine the Petri net model with hierarchical layering. HPN break the complexity of a large model down into several connected submodels that are arranged in a hierarchical structure. This allows to concentrate on a specific system part, without the need to oversee the whole system. Also submodels can be reused with little afford at multiple location in the same system, or even in a different system where similar components are needed.

The increase in size and complexity is not the only challenge modern systems present. Some modern systems, such as flexible manufacturing systems [Chr13], mobile ad-hoc networks [BCGS04] or concurrent systems [MP12], involve dynamic adaptations and are subject to structural change to support their various applications. Such systems can be modeled using reconfigurable Petri nets (RPN) [EHP$^+$07, PEHP08]. RPN are Petri nets that are combined with a rewriting systems to modify the Petri net at run-time. This allows the models structure to change while transitions are being fired. The core of the rewriting system is a set of rules that define the changes that can be made. They provide powerful and intuitive formalisms to model dynamic systems that require both the representation of their processes and of the system changes within one model.

This thesis presents hierarchical reconfigurable Petri nets (HRPN), a model that combines the HPN and RPN models into one, harnessing the advantages of both. Thus allowing a focused design of submodels and their reusability and the ability for dynamic changes at run-time, hence supporting the design of complex dynamic systems. As one of its key properties, the presented HRPN, can be transformed into a non-hierarchical RPN with equal behavior through a process called the flattening process. This has the advantage that only the consistency of the flattening process has to be proven to guarantee the correctness of the model, since then the HRPN model can rely on the correctness of the already proven RPN model. Furthermore, since the behavior of the HRPN and its flattened RPN representation are the same, analyzing and verifying the HRPN can be done through its flattened representation and then requires no more effort than its counter part with no hierarchy.

The aim of this thesis is to first describe a hierarchical reconfigurable Petri net model with formal definitions of net model and flattening process and then to demonstrate and evaluate the model by integrating it into a Petri net modeling tool called ReConNet. ReConNet [PEOH12] is a tool that provides the capabilities to model and simulate reconfigurable Petri nets. For the demonstration and evaluation a use case for HRPN in the form of a flexible manufacturing system is used.

This work is structured as follows: First Chapter 2 provides an overview of relevant background knowledge. The hierarchical reconfigurable Petri net models concept and detailed formal definitions are presented in Chapter 3. Chapter 4 deals with the integration of the HRPN model into ReConNet. It describes the additions and changes to ReConNets program structure and details the implemented flattening algorithm. Then Chapter 5 presents a flexible manufacturing system as an evaluation example. With the FMS example the usefulness of HRPN for flexible systems and the added HRPN modeling capabilities of ReConNet are evaluated. In Chapter 6 an outlook on possible future advancements is given and finally Chapter 7 completes this thesis with a conclusion.

2. Background

In this chapter an overview of relevant background knowledge is provided. In Section 2.1 base concepts of hierarchical Petri nets are presented followed by an explanation on reconfigurable Petri nets in Section 2.2. Section 2.3 is an introduction to ReConNet. Flexible manufacturing systems are discussed in Section 2.4 and finally Section 2.5 addresses related work.

2.1. Hierarchical Petri Nets

Hierarchical Petri nets (HPN) extend Petri nets by hierarchical layering [JR12]. They use hierarchy to break down the complexity of a large model, by dividing it into a number of submodels. This helps the modeler to concentrate on a specific system part without the need to oversee the whole system. Also submodels can be reused with less afford at multiple location in the same system or even in a different system where similar components are used.

There are two mayor techniques of hierarchical Petri nets. *Transition substitution* substitutes transitions with subnets and *place substitution* substitutes places with subnets.

An other approach for hierarchical layering is the usage of Object-Oriented Petri nets (*OOPN*). Currently there exist various sorts of OOPN, [MK05] presents a survey of the most common ones. In it's basic form an OOPN is composed of a system net and it's tokens. Each token represents either a trivial object (e.g. a number or a string) or an object net instance of some class. Every class consists of an object net and a set of dynamically instantiable method nets which structure the models inner behavior. Tokens move from one place to another calling methods and changing attributes. So each token consist of one instance of the appropriate object net and several concurrently running instances of invoked method nets. [KJZJ08] The purpose is to use object oriented constructs to structure and build the system.

Substitution Transition HPN

A hierarchical Petri net can use *substitution transitions* to implement hierarchy. A substitution transition is a special kind of transition that itself does not fire, instead it contains a subnet that defines the behavior that takes place. Following this basic definition of substitution transitions, different implementations suited for specific purposes are possible, this work utilizes a variant of the substitution transition based HPN presented in [JK09].

Each substitution transition has its own subnet. All places that share an edge with a substitution transition are called the transition's *connecting places* and for each connecting place of the substitution transition there exists a corresponding connecting place in the transition's subnet. Each connecting place of the transition and its corresponding connecting place in the subnet form a *connection set*. Both places in a connection set always have the same tokens on them, so if one or more tokens are added or removed from one place of the set the same tokens are also added or removed from the other place of the set. Through these places tokens enter and leave the subnet. During each simulation step one transition can fire, the firing transition can be from either the main net or any subnet. Any net can contain multiple substitution transitions each instantiating exactly one own subnet. Although multiple substitution transitions may instantiate the same subnet layout, each substitution transition has it's own permanent instance. Subnets may also contain substitution transitions, which results in a deeper hierarchy.

Figure 2.1 shows in the top half a hierarchical net with it's main net MN and a subnet SN. In the main net the substitution transition $st1$ has two connecting places: $p0$ has an edges connecting it to $st1$ and $st1$ has an edge connecting it to the place $p1$. These places can also be found in the subnet as connecting places with edges to and from different transitions. If tokens are added to the place $p0$ via the transition $t1$ these also appear in the subnet. There SN's transition sub_t1 can fire and remove tokens from $p0$ resulting in the removal of the same tokens from $p0$ in MN. All connection sets work in this fashion.

This type of hierarchical Petri net can be *flattened*. A process in which the hierarchy of the Petri net is removed. The result is a non-hierarchical Petri net which behaves exactly like the hierarchical Petri net. The specific process steps of the flattening process can vary for each Petri net type and realization. The substitution transition model with connecting places allows employing a relatively straight forward flattening process. During the flattening process one by one every substitution transition is replaced by its subnet and each of its connection sets merged into a single place. In Figure 2.1 this process is used on a simple hierarchical net to create the flattened net F.

Place Substitution HPN

Place substitution works similar to transition substitution. First a place to substitute has to be chosen, than a subnet is created for said place. Similar to the connecting places of substitution transition HPN, *connecting transitions* are created. Each connecting transition in a subnet corresponds to one ore more connecting transitions in the super net, the net of

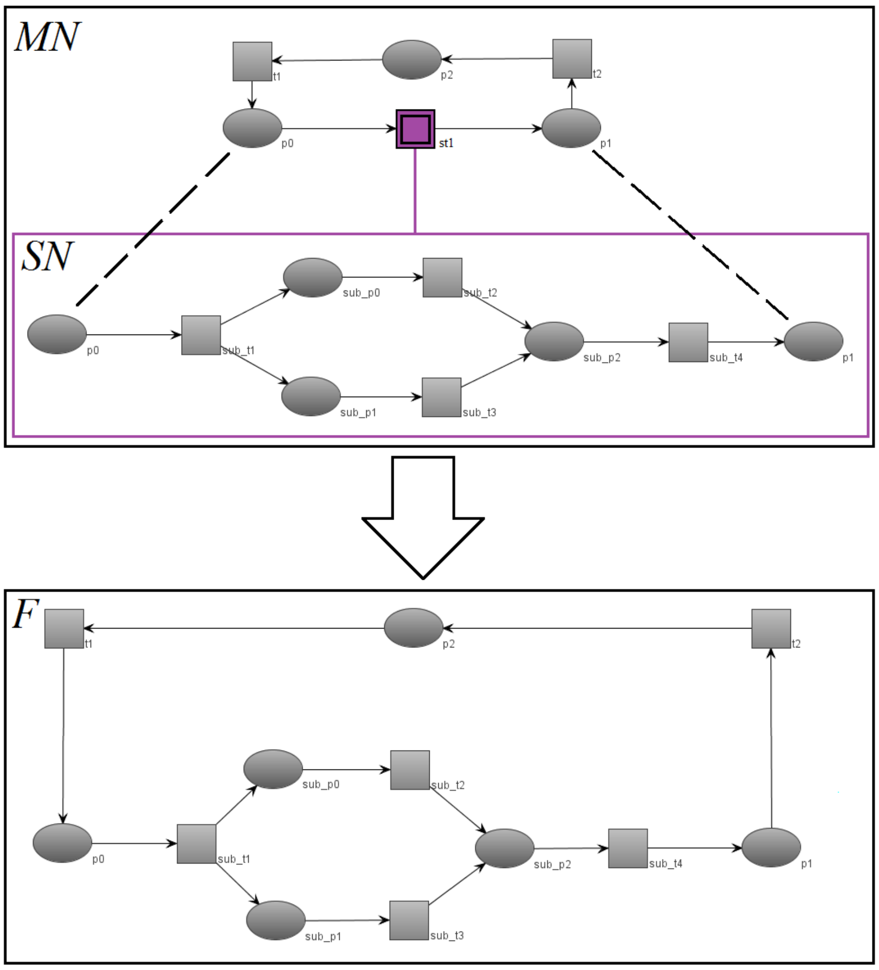


Figure 2.1.: Flattening of a substitution transition.

the substitution place. Whenever one of the connecting transition in the super net fires the corresponding connecting transition in the subnet fires simultaneously. When a connecting transition in the subnet fires one of the corresponding connecting transitions in the super net fires simultaneously. During each simulation step one transition can fire, the firing transition can be from either the main net, any subnet or a pair of connecting transitions. Any net can contain multiple substitution places with each instantiating exactly one own subnet. Although multiple substitution places may instantiate the same subnet layout each place has it's own permanent instance. Subnets may also contain substitution places resulting in a deeper hierarchy.

Figure 2.2 shows a hierarchical net with its main net *SuperPage* and a subnet *Queue*. Four transitions called *Put* and *Init* connect to the substitution place *SubPlace*. *SubPlace* is connected to two transitions called *Get*. All connecting transitions in the super net sharing the same name are combined into one connecting transition in the subnet, resulting in only three connecting transitions in there. When any of the transitions labeled *Put* or *Init* fires, tokens are added to the places *P4* and *P5* or *P5* thus adding tokens to the subnet. If the subnets *Get* transition fires one of the main nets *Get* transitions fires and adds a token to the place *P3*. This way tokens enter and leave the subnet.

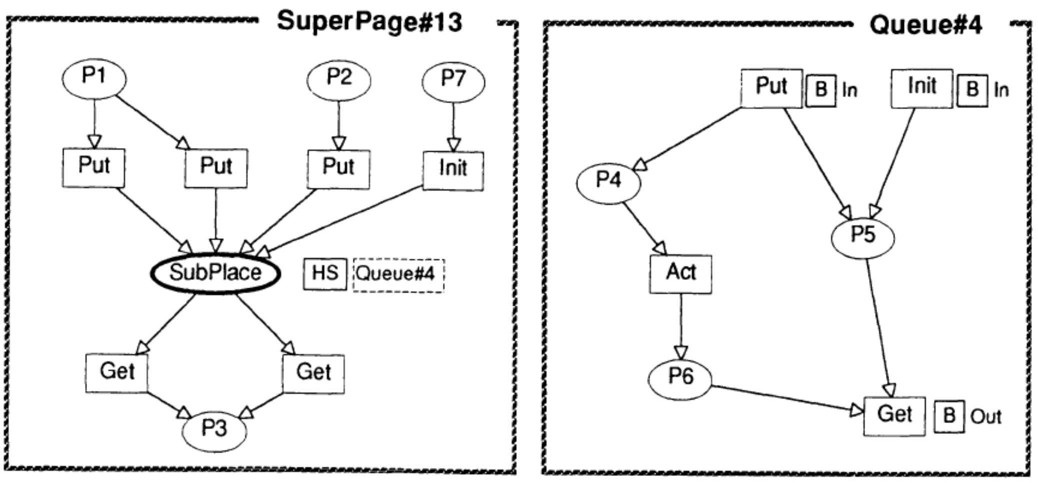


Figure 2.2.: Place substitution example [JR12].

2.2. Reconfigurable Petri Nets

Reconfigurable Petri nets extend normal Petri nets to include the ability for dynamic change. This is achieved through the use of a rewriting system in the form of rules for the transformation of nets [EHP$^+$07, PEHP08]. This allows the modification of the net's structure at run time, which can be used in the modeling of dynamic reconfigurable hardware like FPGAs or flexible manufacturing systems. Such a reconfigurable Petri net enables two kinds change:

- a change of state accomplished through the firing of net transitions
- a change of process attained through the use of the rule based rewriting system.

A reconfigurable Petri net is defined as a marked Petri net N,

Definition 2.1 (Marked Petri Net) *A marked Petri net N is described as a tuple,*

$$N = (P, T, pre, post, M_0)$$

where P is a set of places and T a set of transitions. The pre- and post-domain functions $pre, post : T \to P^{\oplus}$ describe pre- and post-conditions for all transitions. The pre-conditions describe how many tokens are required for the firing of a transition while the post-conditions describe the placement of tokens during the firing of a transition. $M_0 \in P^{\oplus}$ is a set of tokens and defines the initial marking of the net. $P^{\oplus}$ is the free commutative monoid over P [JLL07, MM88].

and a set of transformation rules $\mathcal{R}$.

Definition 2.2 (Transformation Rules) *A transformation rule $r \in \mathcal{R}$ is defined by three nets L, K and R and their strict net morphisms [Pad12] as*

$$r = (L \leftarrow K \rightarrow R)$$

where L is the rule's left-hand side, that is to be located in the net N, and K is an interface between L and R. R is the right-hand side, which is inserted into N. An occurrence morphism $o : L \to N$ is required to identify the relevant parts of the left-hand side L in N.

The basic idea of transformation rules is: when a rule r is applied to a net N an occurrence o of its L net is found in N and gets replaced by R resulting in a new net M. Adding such a set of rules to a marked Petri net forms a reconfigurable Petri net RN.

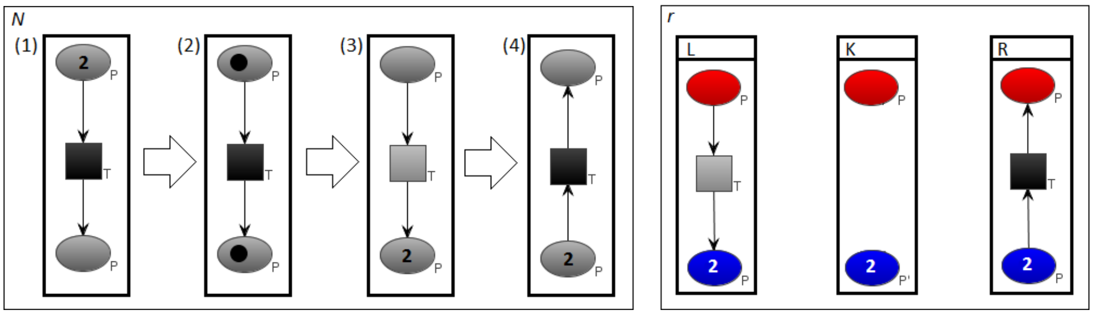
Definition 2.3 (Reconfigurable Petri Net) *A reconfigurable Petri net $RN = (N, \mathcal{R})$ is composed of a marked Petri net N and a set of transformation rules $\mathcal{R}$.*

A reconfigurable Petri net can either fire an activated transition or execute a transformation step $N \overset{(r,o)}{\Longrightarrow} M$. Figure 2.3 illustrates the transformation of a net using two push-out complements (1) and (2). This is possible because Petri nets can be proven to be an $\mathcal{M}$-adhesive transformation category [Pad15, EEPT06].

An example for this process is displayed in Figure 2.4. It shows in (a) a reconfigurable Petri net N and in (b) it's rule r, with r's nets L, K and R. N consists of two places, two tokens and one transition which is black when activated. When rule r is executed the net's arcs are inverted. In it's initial state (1) the rule r is not executable because there is no match to the rules L net. L specifies that at least two tokens are needed on the a place named P where the edges lead to, which is at the bottom. So only the transition T can fire. After transition T firing twice state (3) is reached. In this state two tokens are located on the bottom place P and

$$\begin{tikzcd} L \arrow[d, "o"'] \arrow[dr, phantom, "(1)"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d] \\ N & D \arrow[l] \arrow[r] & M \end{tikzcd}$$

Figure 2.3.: Transformation of a net [Pad15].



(a) State sequence of net N (b) Exemplary rule r

Figure 2.4.: Example Petri net N and rule r.

r can be executed inverting the arc directions resulting in state (4). If in state (3) there would have been an additional token on the upper place P either the transition T could have fired or r could have been executed. In state (4) r is no longer executable because the edges now lead to the upper place P, so only the transition T can fire. State (4) is very similar to state (1) and after T firing twice r would be executable once again.

Reconfigurable Petri net rules can be extended using negative application conditions (NAC). NAC in reconfigurable Petri nets have been introduced in [RPL$^+$08] and provide the possibility to forbid certain rule applications. They restrict the application of a rule by forbidding a certain structure to be present before or after applying a rule in a certain context. Rules with NACs have an additional set of nets NC_i, denoting the forbidden contexts. Formally, a rule is applicable only if its match m cannot be extended to a match m_i of NC_i [PH15].

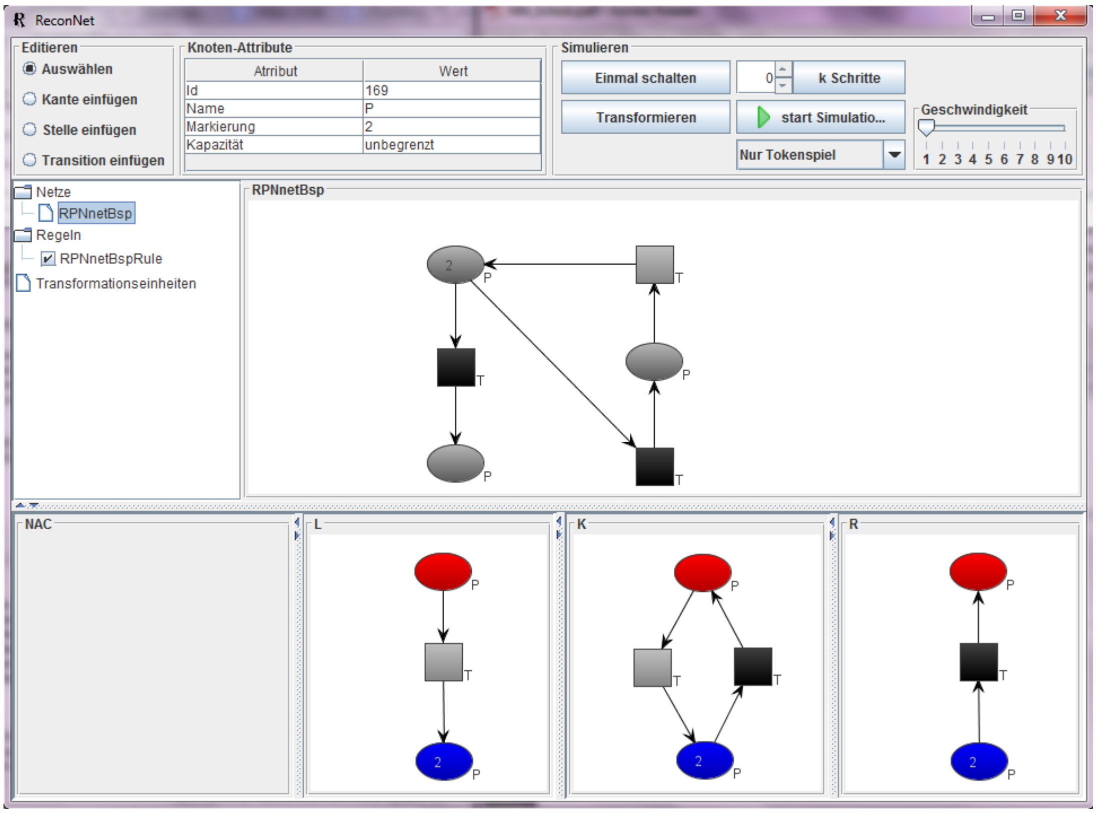
2.3. ReConNet

To model and simulate the capabilities of reconfigurable Petri nets a tool called ReConNet [PEOH12] can be used. It is completely implemented in Java 6 and provides a graphical user interface, as can be seen in figure 2.5. In the left center Petri nets and rules can be managed.

Petri nets are displayed in the center, rules with their nets L, K and R at the bottom. The top part of ReConNet's GUI shows from the left to the right: modeling tool selection, node attributes and simulation/transformation tools.

Instead of a span approach for rules: $r \in \mathcal{R} = L \leftarrow K \rightarrow R$, ReConNet uses a co-span approach, as described in [EHP09], for it's rules: $r \in \mathcal{R} = L \rightarrow K \leftarrow R$. This results in the gluing net K being a union of L and R that contains all places, transitions and edges that are in at least one of them. The span and co-span approaches result both in the same net after the transformation.

The bases for ReConNet are decorated PT nets [Pad12]. Decorated PT nets extend PT nets with additional decorations like names for places and transitions, capacities and transition labels that can change during the firing of a transition. The additional labels allow further coordination of transition firing and rule application. This provides a tool to control the application of rules while preserving the nets behavior.



Atrribut	Wert
Id	169
Name	P
Markierung	2
Kapazität	unbegrenzt

Figure 2.5.: ReConNets graphical user interface.

2.4. Flexible Manufacturing Systems

A flexible manufacturing system (FMS) [KV10] is a group of computer numerically controlled (CNC) machines that are connected, via loading and unloading stations, to an automated transport system. It seeks the middle ground between a standalone computer numerical controlled machine that is capable of producing a variety of products and a transfer line, which consist of a predetermined sequence of machines.

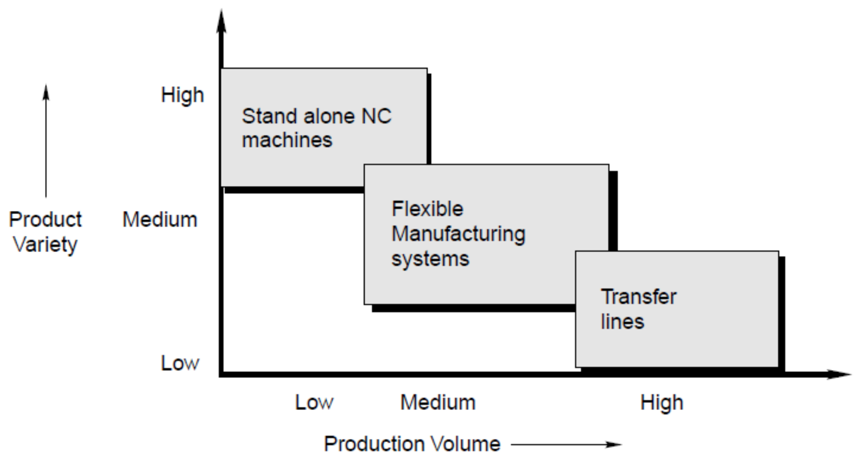


Figure 2.6.: FMS in relation to other production solutions [LWL+06].

In a FMS the transport system carries work to and from the machines by means of an automated material handling and storage system. The system parts are central controlled and able to respond to changed conditions. This automated production system is capable of processing and manufacturing a variety of part types at a rapid speed. [Tet90] A FMS usually consist of three main parts:

- CNC machine units
- an automated material handling and storage system
- a computerized planning and control system

Figure 2.7 shows an exemplary FMS that uses a *loop layout* for its transport system. All raw materials and produce are carried by the same transport system and the control system decides which items are loaded or unloaded at each intersection and chooses what a CNC unit does with the materials it receives. This way any combination of process steps capable by the CNC units can be accomplished on the fly.

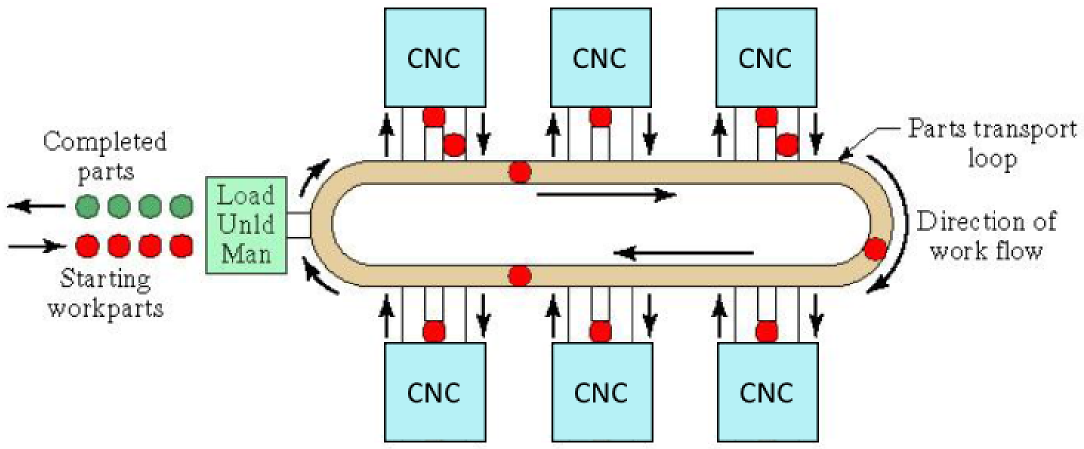


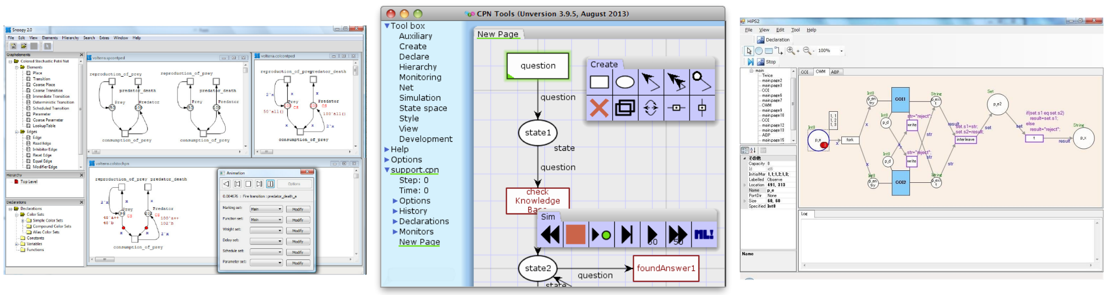
Figure 2.7.: Basic loop layout of a FMS [LGB18].

In the center of a loop layout FMS stands a circular transport systems on which parts loop around until they reach their destination. Depending on the requirements and main focus of a use case, different FMS and layout concepts are possible. [MRA10] and [Tet90] elaborate further on different techniques of manufacturing systems and possible FMS layouts.

2.5. Related Work

A number of tools similar to ReConNet exist. *Snoopy* [HHL+12] is one of these tools, it is a unifying Petri net framework with a graphical user interface. It allows the modeling and simulation of colored and uncolored Petri nets of different classes, it also supports analytic tools and the hierarchical structuring of models. *CPN tools* [RWL+03] is another tool for the modeling and simulation of colored Petri nets. Using a graphic user interface CPN tools features syntax checking, code generation and state space analysis. The *HiPS* tool [HiP17] developed at the Department of Computer Science and Engineering, Shinshu University is a tool written in C# and also employs a graphical user interface. HiPS is a platform for design and simulation of hierarchical Petri nets. It also provides functions of static and dynamic net analysis. While all of these tools support the design of hierarchical Petri nets each lacks ReConNet's core feature the aspect of reconfigurability.

There are many use cases for hierarchical Petri nets, one can be found in [SCDB14]. There hierarchical colored Petri nets are used to model the French railway interlocking system *RIS* for formal verification and logic evaluation. The RIS system is responsible for the safe routing of trains. Detailed verifications and evaluations are mandatory before deploying an RIS, since it



(a) Snoopy (b) CPN tools (c) HiPS

Figure 2.8.: Petri net tools.

is a safety critical system. The paper describes how the signaling control and the railway road layout are specified and constructed into a colored hierarchical Petri net. In [ZZ09] hierarchical colored Petri nets are used to model the production process of a cold rolled steel mill. For this a crude description of the entire running process of the system is given at the main net, and the more detailed behaviors are specified in the subnets. It is shown that the design is highly consistent with real production, improving the development efficiency for production planning and scheduling.

The utilization of reconfigurable Petri nets for flexible manufacturing systems is prevalent, in [TPCS12] the concept of reconfigurable finite capacity Petri nets applied to a flexible manufacturing system model is explored. This concept is then used to model a simplified scenario involving a FMS which is able to assemble several products.

ReConNet core feature is the ability to model and simulate reconfigurable Petri nets, although in the department of other reconfigurable Petri net tools there is not much to find there exist a number of graph tools that allow transformations. General purpose graph tools like *GROOVE* [Ren03] and *AGG* [Tae99] allow the design, simulation and transformation of graphs. By defining the basic rules of a Petri net, these tools can be used to produce Petri net graphs and allow transformations.

This thesis is based on two previous works. The previous work [Lor17a] investigates different hierarchical Petri net types for the use in hierarchical reconfigurable Petri nets. Besides hierarchical Petri nets based on transition substitution, nets based on place substitution and Object-Oriented Petri nets (*OOPN*) are considered. The follow up [Lor17b] presents a concept for hierarchical reconfigurable Petri nets (HRPN) and provides formal definitions and proves of the hierarchy and reconfiguration properties. In [LP18] a variation of the HRPN concept is presented, it uses Petri nets with labels and subtyping of labels to allow global transformation rules that can be applied to any level of the hierarchical net.

3. Concept for Hierarchical Reconfigurable Petri Nets

The concept of a hierarchical reconfigurable Petri net (HRPN) can be divided into two major areas: the hierarchical model that is used to realize the hierarchical aspects and the transformation rules that are responsible for the reconfigurability of the Petri net.

The concept presented here is based upon the concept described in [Lor17b], while Section 3.1 provides the general concept choices for hierarchy and reconfiguration, Section 3.2 and Section 3.3 go into further detail and provide formal definitions for the structure of the HRPN and the transformation rules.

3.1. General Concept

This section provides the basic designs used in the HRPN concept with additional remarks on the reasoning of the conceptual choices and special properties of HRPN. Since the intend is to integrate hierarchy into the reconfigurable Petri net model that is used by the ReConNet tool, the presented HRPN concept is designed with its usability for ReConNet in mind.

3.1.1. Hierarchical Model

For the hierarchical model the use of a model based on *substitution transitions* is chosen. In particular a model that can be flattened into a non-hierarchical Petri net model of equal behavior as it is described in Section 2.1. The ability to flatten the net into a non-hierarchical Petri net of equal behavior is useful during the implementation process and allows the utilization of methods of validation and verification that are commonly used on non-hierarchical nets. Substitution transitions are also chosen because many related tools like *snoopy* [HHL+12], *CPN tools* [RWL+03] or *HiPS* [HiP17] use substitution transitions, which allows for if not compatibility, at least comparability. Also the model aims to support the usage of hierarchical Petri nets for as many applications as possible and a majority of papers concerned with hierarchical Petri nets use substitution transitions to achieve their goal, [SCDB14] uses hierarchical colored Petri nets to model the French railway interlocking system and [ZZ09] used hierarchical colored Petri nets to improve the planning and scheduling efficiency of a cold rolled steel mill.

3.1.2. Transformation Rule Concepts

As Section 2.2 elaborated, reconfigurable Petri nets use transformation rules to reconfigure themselves. The combination with a hierarchical model presents a certain challenge. With the chosen hierarchical model three different transformation rule concepts were compiled: *global rules*, *local rules* and *layer based rules*.

Global Rules

A global transformation rule is a general rule that may be applied in any net on any level of the whole hierarchical net. So occurrences can be in the main net, its subnets, all their subnets and so forth.

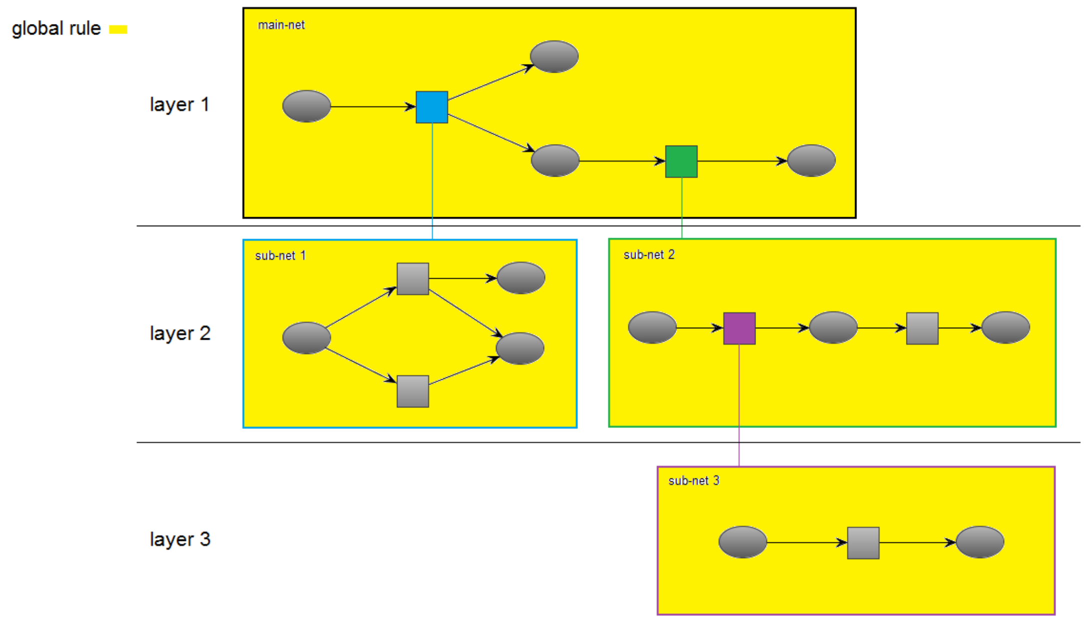


Figure 3.1.: Application of global rules to a hierarchical Petri net.

Local Rules

A local rule applied to a (sub-)net allows occurrences to only be found in that specific net. Local rules enable the dynamic modification of a specific part of a hierarchical net. The advantage is that rules can be created without the knowledge of other parts of the hierarchical net.

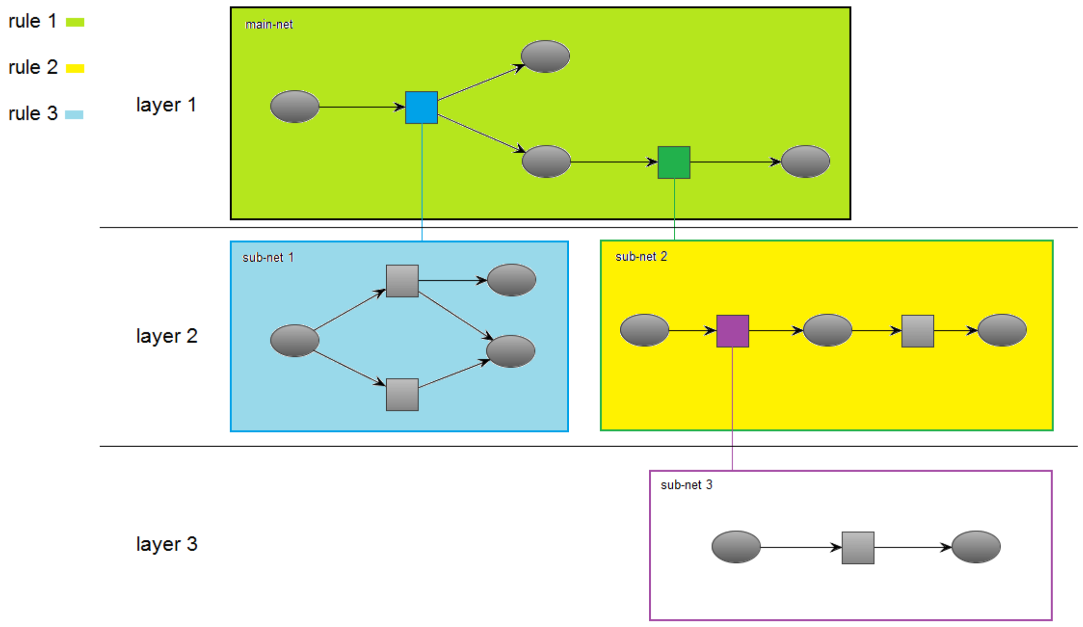


Figure 3.2.: Application of local rules to a hierarchical Petri net.

Of course a local rule's design can be reused an applied to a different (sub-)net as well thus effectively enabling the modification of a set of (sub-)nets.

Layer Based Rules

Layer based rules are applied to nets relative to the main net. A layer based rule will only be applied to nets of its designated layer. While in layer 1 there is only the main net, layer 2 composes of all subnets that are derived directly from the main net. Layer 3 composes of all their direct child nets and so on.

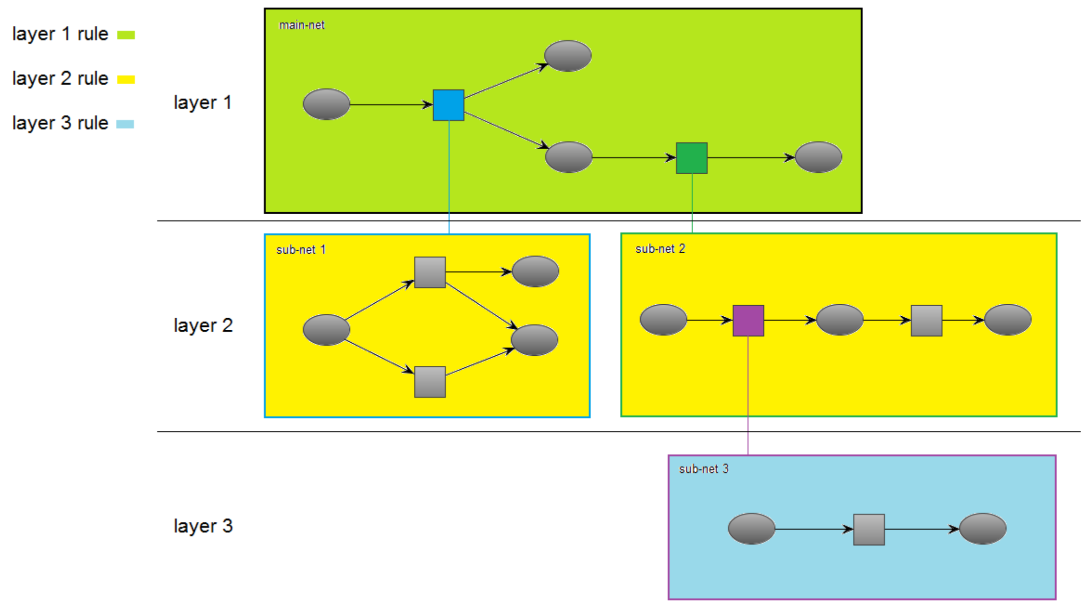


Figure 3.3.: Application of layer based rules to a hierarchical Petri net.

Special Properties of Hierarchical Reconfigurable Petri Nets

The hierarchical net is specifically designed to allow black-box behavior of subnets. In a hierarchical reconfigurable Petri net it must be ensured that the internal workings of a subnet do not change by applying a transformation rule to its super net. For this transformations may not span across hierarchical borders.

Figure 3.4 shows a net with one subnet. To the net a global rule is applied as shown in figure 3.5. This rule adds two places and can be applied at multiple locations.

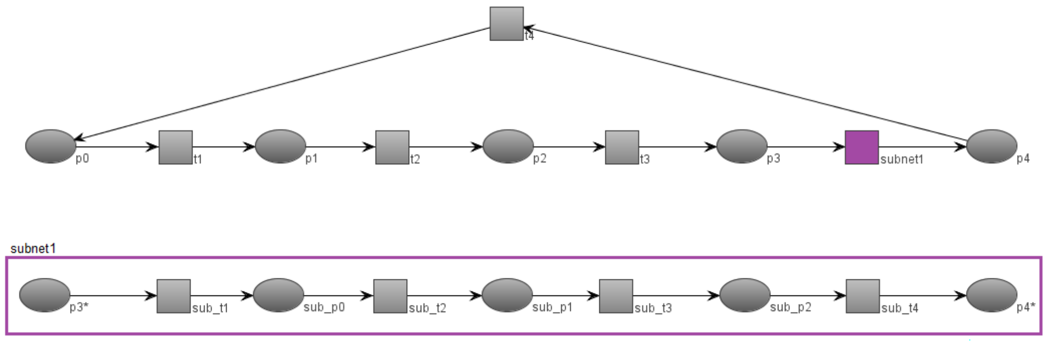


Figure 3.4.: A basic hierarchical reconfigurable Petri net.

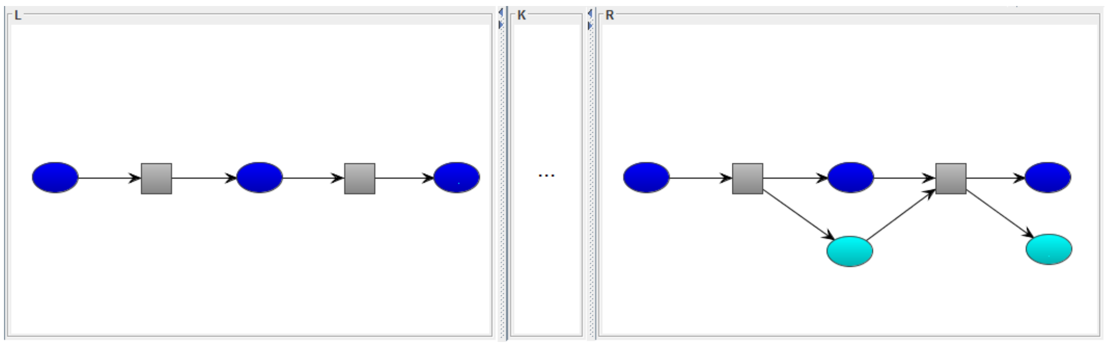


Figure 3.5.: Rule for the hierarchical reconfigurable Petri net.

In figure 3.6 three matched occurrences are highlighted. The addition of the places marked in green and yellow is straight forward. The rule application creating the green places only effects the main net and the application creating the yellow places effects only the subnet. The addition of the places marked in red would be possible in a non-hierarchical net. However the hierarchical layout excludes the rule application across hierarchy borders. Thus the rule must not be applied under such constraints.

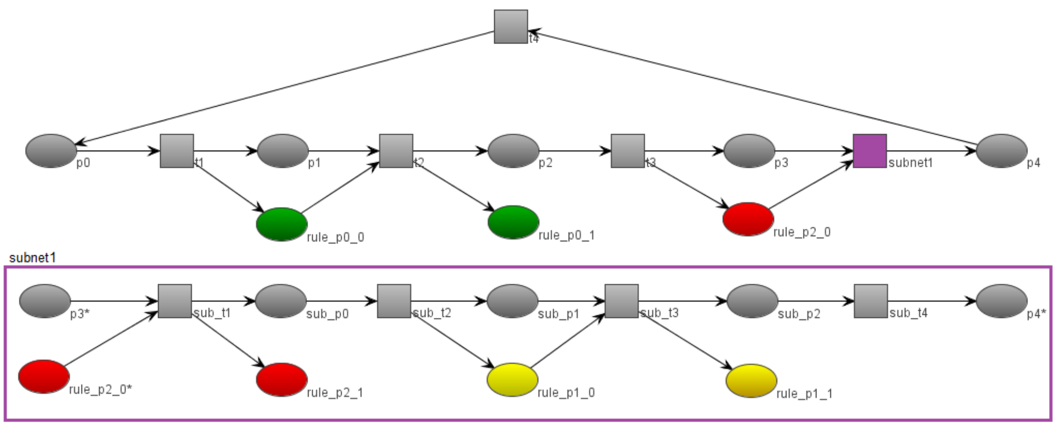


Figure 3.6.: Considered rule application locations.

Since for the hierarchical Petri net model an approach is chosen that can be flattened into a non-hierarchical Petri net model of equal behavior, the limitations to transformation rules imposed by the hierarchical borders must be preserved in the flattened net. As a consequence a more complex flattening process, than for a normal hierarchical Petri net, is necessary. This is due to the fact that in the process of flattening a normal hierarchical Petri net all information on where the hierarchical borders were located are lost. These information are needed for the correct application of transformation rules in the flattened net, so that the behavior of the net is not changed by flattening it. This more complex flattening process assures that a transformed net is always the same in both the cases of: first applying a rule and than flattening it and first flattening it and than applying the rule. Section 3.3 details how this can be achieved.

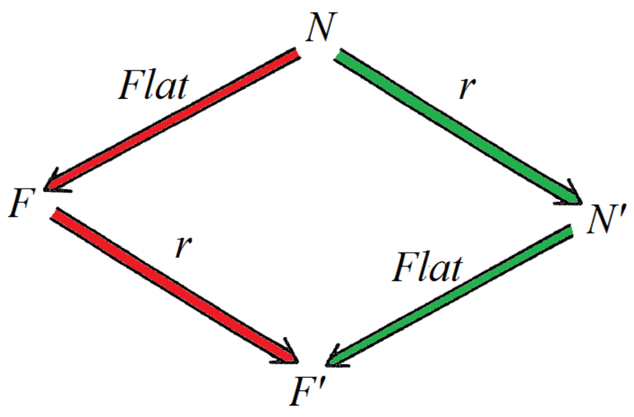


Figure 3.7.: Flattening and transformation of a hierarchical reconfigurable net.

3.2. Hierarchy and Flattening of HRPN

Following the concept of Chapter 3.1, for a hierarchical reconfigurable Petri net (HRPN), this chapter provides in Section 3.2.1 a formal definition for HRPNs. In Section 3.2.2 the flattening process, that is used to obtain the non-hierarchical representation of a HRPN, is defined.

3.2.1. The Hierarchical Reconfigurable Petri net

The HRPN is mainly defined by it's flattening into a non-hierarchical reconfigurable net. The HRPN consists of a reconfigurable net RN and a set of substitution rules SR. To define this properly, first the surrounding net $Net(t)$ of a transition t is being defined as follows:

Definition 3.1 (Net(t)) *The net of t, $Net(t)$, is the net surrounding a transition t. With the tuple $Net = (P, T, pre, post, p_{name}, t_{name})$ describing a net, then $Net(t)$ is defined as* $Net(t) = ({}^\bullet t \cup t^\bullet, t, pre_{|t}, post_{|t}, p_{name_{|{}^\bullet t \cup t^\bullet}}, t_{name_{|t}})$.

With this HRPN can be formally defined:

Definition 3.2 (Hierarchical Reconfigurable Petri Net) *A hierarchical reconfigurable net $HN = (RN, A, SR)$ is given by a reconfigurable net $RN = (N, \mathcal{R}^N)$, a name space $A = (A_P^N, A_T^N)$ and a set of substitution rules SR, so that*
$RN = (P, T, pre, post, p_{name}, t_{name}, M, \mathcal{R})$ *with:*

- $A_{cP} \subseteq A_P^N$: *the name space of connecting places and* $A_{sT} \subseteq A_T^N$ *the name space of substitution transitions.*
- P : *a set of places that also contains connecting places* $cP \subseteq P$.
- T : *a set of transitions that also contains substitution transitions* $sT \subseteq T$.
- $pre : T \to P^{\oplus}$ *a function used for all pre-domains of each transition.*
- $post : T \to P^{\oplus}$ *a function used for all post-domains of each transition.*
- $p_{name} P \to A_P^N$: *a naming function for places with* $p_{name}(cP) \subseteq A_{cP}$ *and* $p_{name}(P \backslash cP) \subseteq A_P^N \backslash A_{cP}$.
- $t_{name} T \to A_T^N$: *a naming function for transitions with* $t_{name}(sT) \subseteq A_{sT}$ *and* $t_{name}(T \backslash sT) \subseteq A_T^N \backslash A_{sT}$.

- *M : a set of tokens by $M \in P^{\oplus}$.*
- *$\mathcal{R}^N$: a set of transformation rules over $(A_P^N, A_T^N \setminus A_{sT})$.*

SR is a set of substitution rules, together with a mapping of substitution transitions to substitution rules: $sT \to SR$ so that $sr(st) = ST(st) \leftarrow CP(st) \to SN(st)$ with $sr \in SR$, $st \in sT$ and

- *$ST(st) = Net(st)$*
- *$CP(st) = ({}^{\bullet}st \cup st^{\bullet}, \emptyset, \emptyset, \emptyset, p_{name_{|{}^{\bullet}st \cup st^{\bullet}}}, \emptyset)$*
- *$SN(st)$ being a hierarchical net $SN(st) = (\widehat{RN_{st}}, \widehat{A_{st}}, \widehat{SR_{st}})$ with $A_{cP} \subseteq \widehat{A_{st}}$.*

Figure 3.8 shows an example for a very basic substitution rules.

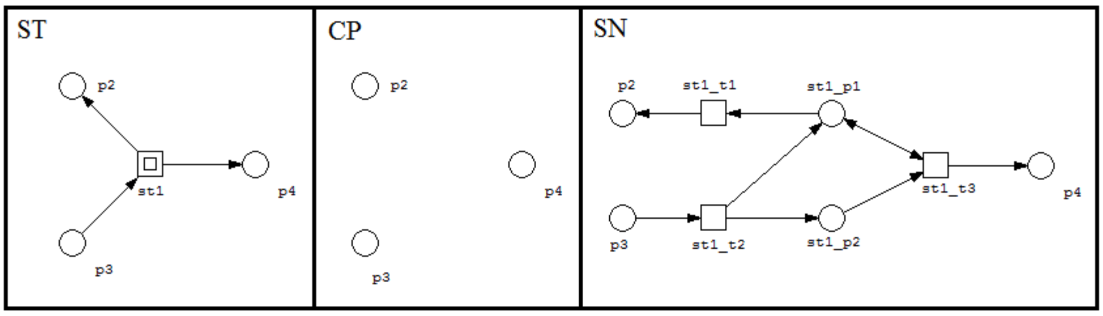


Figure 3.8.: An exemplary substitution rule.

3.2.2. Hierarchical Flattening

The non-hierarchical representation of a hierarchical reconfigurable Petri net is a reconfigurable Petri net. Through the *flattening* process this non-hierarchical (*flattened*) Petri net representation can be obtained. The basics of the flattening process were previously described in Section 2.1.

In [JK09] Chapter 5 it states that for the flattening of a hierarchical net that uses substitution transitions each substitution transition must be removed and its subnet inserted into the super net by fusing the connecting places. Figure 3.9 shows a hierarchical Petri net with its main net *MN* and one subnet *SN* housed in the substitution transition $st1$.

Through the removal of the substitution transition $st1$ from *MN* and inserting the subnet *SN*, by fusing the SNs and MNs connecting places labeled $p2$, $p3$ and $p4$ respectively, the flat representation of the hierarchical Petri net displayed in Figure 3.10 can be acquired.

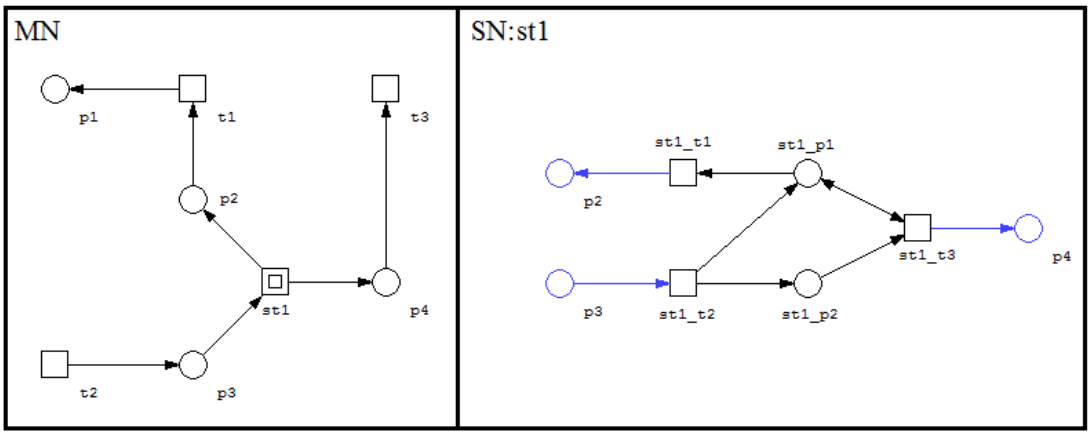


Figure 3.9.: A hierarchical Petri net.

This process can also be modeled using the substitution rules from Definition 3.2:

Definition 3.3 (Substitutions) *For all $sr \in SR$ and for all injective occurrences o of sr there exists exactly one substitution $s = \{(sr, o) | (sr, o)$ is applicable$\}$ for every substitution transition st. These substitutions are collected in a set $s \in S$.*

Definition 3.4 (Flattening Process) *The flattening is defined for an hierarchical net $HN = (RN, A, SR)$ given by a reconfigurable net $RN = (N, \mathcal{R}^N)$, a name space $A = (A_P^N, A_T^N)$ and a set of substitution rules SR as given in Def. 3.2 recursively by*

1. *$flat((N, \mathcal{R}), A, SR) = (N, \mathcal{R})$ if $sT = \emptyset$*
2. *$flat(RN, A, SR) = flat(FLAT((N, \overline{\mathcal{R}}), SR), \overline{A}, \overline{SR})$ with*
   - $\overline{A} = \biguplus_{st \in sT} (\widehat{A_{st}} \setminus A_{cP}) \uplus A_{cP}$
   - $\overline{\mathcal{R}} = \biguplus_{st \in sT} \widehat{\mathcal{R}_{st}}$
   - $\overline{SR} = \biguplus_{st \in sT} \widehat{SR_{st}}$

with $FLAT((N, \overline{\mathcal{R}}), SR) = (\overline{N}, \overline{\mathcal{R}})$ by applying each substitution s once, $s \in S$ from Definition 3.3.

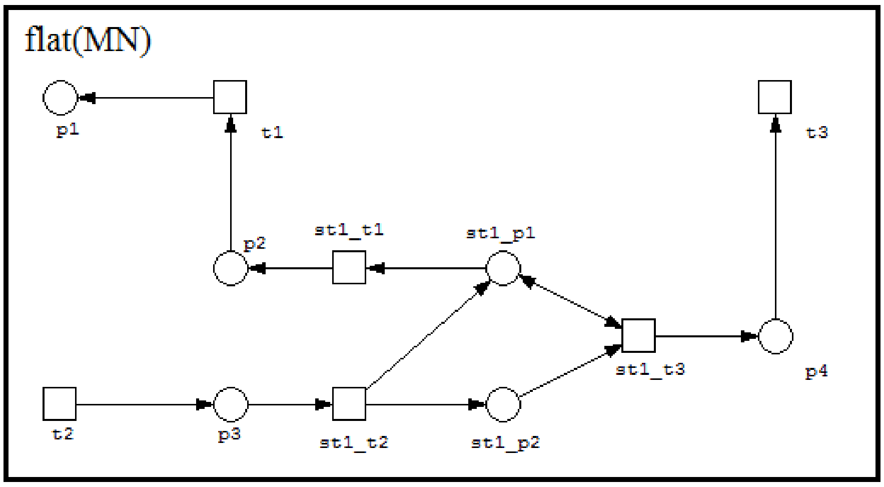


Figure 3.10.: The flat representation of a hierarchical Petri net MN.

Definition 3.5 (Disjoint Union) *Given set A and B then $A \uplus B$ is given by the coproduct construction, so that for any $f : A \to C$ and $B \to C$ there is a unique $h : A \uplus B \to C$ with $h \circ incl_A = f$ and $h \circ incl_B = g$, [EEPT06] as in the diagram below:*

$$\begin{tikzcd}
A \arrow[dr, "incl_A"'] \arrow[drr, bend left, "f"] & & \\
& A \uplus B \arrow[r, "h"] & C \\
B \arrow[ur, "incl_B"] \arrow[urr, bend right, "g"'] & &
\end{tikzcd}$$

When applied the flattening process from Definition 3.4 transforms a hierarchical net into a non-hierarchical one.

Theorem 3.1 (The Flattening Process Produces a Well-defined Net F) *Any possible transformation sequence during the flattening process results in the same net F that is well-defined up to isomorphism.*

Because all substitution rules can be proven to be pair-wise independent [Lor17b] Theorem 3.1 can also be proven:

Proof Sketch 3.1 (The Flattening Process Produces a Well-defined Net F) *With all $s \in S$ being mutually independent, [RE97] states all the transformation sequences $HN \overset{*}{\Rightarrow} F$ are* *equivalent and there exists a parallel transformation sequence* $HN \xrightarrow{\sum_{s \in S} s} F$. *Also it is always possible to construct such a parallel transformation sequence and that sequence is unique up to isomorphism. So with all sequences being equivalent the resulting* F *is well-defined up to isomorphism.*

Flattening as Transformation Unit

The flattening process can also be realized as transformation unit, transformation units encapsulate rules and control conditions that regulate the application of rules to graphs including the specification of initial and terminal graphs [KKR08]. The transformation unit used for the flattening process uses the as long as possible operator (!) : $HN \overset{\text{sr!}}{\Rightarrow} F$ with injective occurrences. For this transformation unit an applicable substitution rule sr with an occurrence is randomly picked and applied, this step is repeated until there no longer exists a $sr \in HN$ with an occurrence. The approach with a transformation unit also produces a well-defined flat net F [Lor17b].

3.3. Transformation Rule Concept

As mentioned in Section 3.1.2 transformation rules in general and local rules in particular need to be limited so that they do not induce changes across hierarchical borders. To prevent transformation rules from altering hierarchical borders in a HRPN special restrictions on transformation rules are defined in Section 3.3.1. These restrictions also ensure that a HRPN and the flat net obtained through the flattening process are of equal behavior both in firing transitions and applying transformation rules, Sections 3.3.2 elaborates on this equality of their behavior.

3.3.1. Transformation Rules in HRPN

In Chapter 2 three possible transformation rule variants were presented: *local rules*, *global rules* and *layer based rules*. Both global rules and layer based rules can be modeled by a systematical usage of local rules, so for the purpose of simplicity all further considerations are made with local rules in mind only.

To preserve the designed hierarchical layer borders, no transformation rule may effect more than one (sub-)net. To realize this, two restrictions are imposed on transformation rules. Firstly substitution transitions cannot be part of a transformation rule, i.e. only atomic nets are allowed as L, K and R nets of a transformation rule.

Definition 3.6 (Atomic Net (AN)) *An atomic net AN is a non-hierarchical net and can be defined using the notation of Definition 3.2 as $AN = (RN, \emptyset)$ with $sT = \emptyset$.*

Definition 3.7 (Transformation Rule Restriction 1) *For all transformation rules $r \in \mathcal{R}$ it applies: The nets L, K and R of $r = L \leftarrow K \rightarrow R$ are atomic nets from Definition 3.6.*

Secondly connecting places may not be deleted or added by a transformation rule, but they can be part of one. So for all transformation rules $r = L \leftarrow K \rightarrow R$ it applies: If a connecting place is part of the left-hand side L or the right-hand side R of a transformation rule r than the connecting place is also part of r's interface K and thus part of all of r's nets L, K and R.

Definition 3.8 (Transformation Rule Restriction 2) *For all connecting places $p \in cP$ in a transformation rule $r = L \leftarrow K \rightarrow R$ it applies that from $p \in L$ or $p \in R$ it follows $p \in K$. With $pre(t) \subseteq cP^{\oplus}$ and $post(t) \subseteq cP^{\oplus}$ if $t \in sT$.*

3.3.2. Equality of Behavior

The HPN model can always be flattened into an equivalent non-hierarchical model with the same behavior [JR12]. This is also true for a hierarchical reconfigurable Petri net. The behavior of a HRPN model is composed of the simulation behavior and the transformation behavior. While the simulation behavior is defined by the activation of transitions and the movement of tokens, the transformation behavior is defined through the application of transformation rules and how they change the net and its behavior.

To make certain that the proposed HRPN model suffices these behavioral constrains this section elaborates on the model's conditions. Section 3.3.2 gives insight on the model's simulation behavior without the interference of transformation rules and Section 3.3.2 elaborates on the transformation behavior both before and after the application of the flattening process.

Simulation Behavior

Without the reconfiguration rules a hierarchical reconfigurable Petri net (HRPN) simply is a hierarchical Petri net (HPN). The HRPN uses a colored HPN presented in [JK09] and [JR12] as a basis. The used HPN model is directly comparable to the HPN model used by Jensen and Kristensen. Chapter 5 of [JK09] elaborates on this HPN and its Section 5.6 in particular on the behavioral equality of the HPN and its flat representation acquired through the flattening process, which there is called $unfolding$. By using this model it is ensured that the simulation behavior of the HRPN and its non-hierarchical presentation are the same.

Transformation Behavior

For the transformation behavior of a HRPN and its flattened representation to be the same the application of a rule must be independent from the form of the net. Be it its hierarchical representation N or its flat representation F. Thus the flattening process $Flat$ and any rule r need to be independent from one an other as seen in Figure 3.11 on the left.

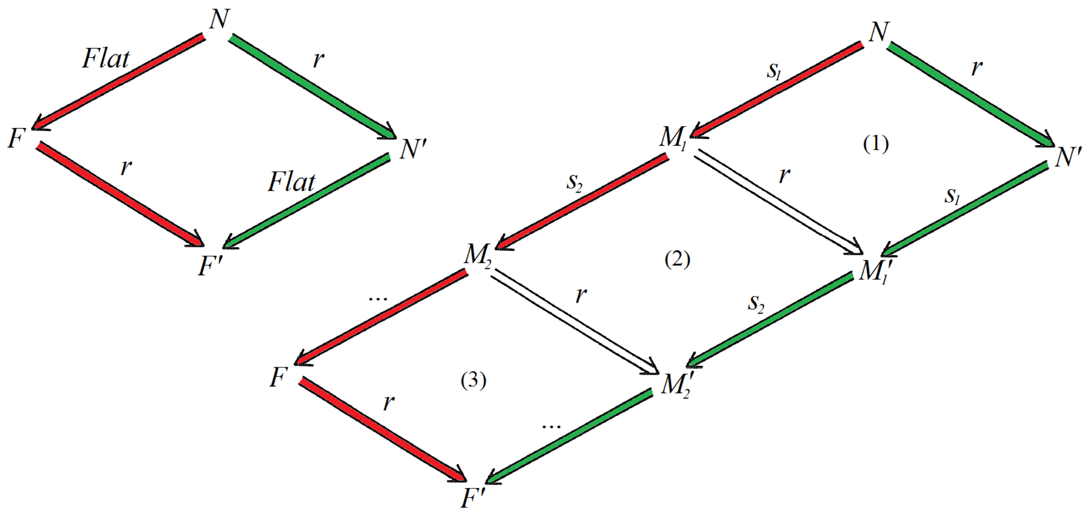


Figure 3.11.: Possible combinations of flattening a net and applying a rule: In red applying a rule to the flat net, in green flattening a net in which the rule was already applied.

Theorem 3.2 (Independence of Transformation and Flattening) *The transformation sequence $HN \xRightarrow{Flat} F \overset{r}{\Rightarrow} F'$ is equivalent to $HN \overset{r}{\Rightarrow} HN' \xRightarrow{Flat} F'$ with HN being a reconfigurable hierarchical net from Definition 3.2, r one of HN's transformation rules and $Flat$ the flattening process of HN from Definition 3.4.*

$$HN \xRightarrow{Flat} F \overset{r}{\Rightarrow} F' \equiv HN \overset{r}{\Rightarrow} HN' \xRightarrow{Flat} F'$$

Because the pairwise independence of any two substitution rules and the independence of any substitution rule and any transformation rule can be proven [Lor17b], theorem 3.2 can also be proven as well:

Proof Sketch 3.2 (Independence of Transformation and Flattening) *All $s \in S$ of HN are pairwise independent, $HN \xRightarrow{Flat} F$ can also be constructed maximum parallel as $HN \xRightarrow{\sum_{s \in S} s} F$ or as a transformation sequence $HN \overset{*}{\Rightarrow} F = HN \overset{s_1}{\Rightarrow} \ldots \overset{s_n}{\Rightarrow} F$ with $n = |S|$ [RE97].*

Since r is mutually independent from all s and since any sequence of sequentially independent transformations can be applied in arbitrary order yielding the same well-defined resulting net [EEPT06], r can be applied before or after all or any s: $HN \overset{*}{\Rightarrow} F \overset{r}{\Rightarrow} F' \equiv HN \overset{r}{\Rightarrow} HN' \overset{*}{\Rightarrow} F'$ and thus

$$HN \xRightarrow{Flat} F \overset{r}{\Rightarrow} F' \equiv HN \overset{r}{\Rightarrow} HN' \xRightarrow{Flat} F' \tag{3.1}$$

Thereby for a hierarchical reconfigurable net HN it does not matter if one of its transformation rules r is applied before or after the net was flattened to its non-hierarchical representation F, thus the transformation behavior does not change through the flattening of a net and the transformation behavior of a hierarchical net HN and its flat net F are equal.

4. HRPN in ReConNet

Since the concept of hierarchy is new to ReConNet, for the integration of the hierarchical reconfigurable Petri net from Chapter 3 there are three mayor points to consider:

- Simulation of HPN
- Storing and restoring of HRPN
- Application of local transformation rules

This chapter presents an implementation approach for HRPN into ReConNet, first a brief overview of ReConNets program structure and Petri net model is given, so that later the changes and additions for the incorporation of HRPN into ReConNet can be discussed in further detail. Afterwards Section 4.1 deals with the incorporation of the HPN model into ReConNet. In Section 4.2 a solution to the persistence problem of storing and restoring HRPN with ReConNet is presented. Then Section 4.3 and Section 4.4 discuss simulation and realization of local rules in ReConNet. Finally Section 4.5 details ReConNets flattening algorithm.

ReConNet structure

ReConNet is written in Java 6 and it consists of five main components: *gui*, *petrinet*, *persistence*, *transformation* and *engine*. The two packages *exceptions* and *util* add support for the components with custom exceptions and other utensils. An overview of ReConNets program structure with the most significant program parts is displayed in Figure 4.1.

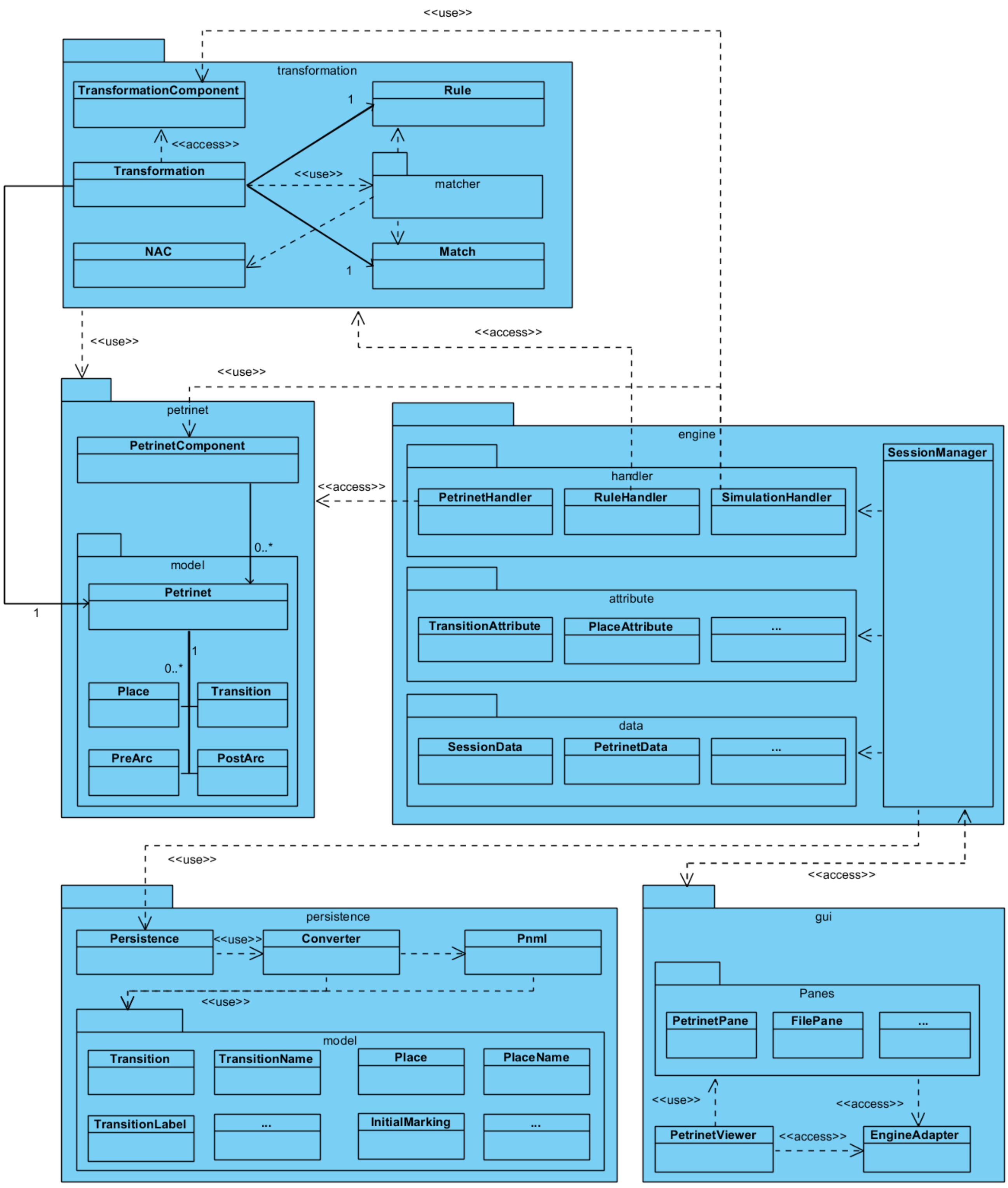


Figure 4.1.: Overview of ReConNets program structure.

The *engine* component operates at ReConNets core and controls the program flow. It holds data (e.g. *SessionData*, *PetrinetData*) , data structures (e.g *TransitionAttribute*, *PlaceAttribute*) and handlers for all other relevant program parts and acts as an intermediate between them. The data objects of the *engine* component contain information about all Petri nets and rules and about the current program session. All information is accessible via its ID so that other program parts only need to hold the IDs of information they require. The handlers *PetrinetHandler* and *RuleHandler* only control the access to the *petrinet* and *transformation* components, while the *SimulationHandler* organizes all the simulation related functionalities, like firing a random activated transition or applying a random rule with an occurrence.

The graphical user interface is managed by the *gui* component. It is divided into six areas (panes), each for a specific purpose: Petri nets, rules, RPN simulation, editing of node or arc properties, tool selection and management of files. Access to and from the *gui* component is realized through the *EngineAdapter* and the *engine* component.

The *transformation* component realizes the creation and application of rules, the component is accessed through its *TransformationComponent* class. For the application of a rule a *Transformation* object is created, it contains a reference to the rule that is to be applied, the Petrinet that the rule is to be applied to and an a *Match*, found by the *matcher* subcomponent, that represents the occurrence of the rule in the Petri net. The *Transformation* object also holds the tools necessary to apply the matched rule to the Petri net.

The *petrinet* component manages ReConNet's Petri nets and Petri net related functionalities like creating Petri nets, editing Petri nets, activating transitions and firing transitions as well as the Petri net model structure itself. The model consists of *Places*, *Transitions* and directed arcs. The arcs are divided into *PreArcs* and *PostArcs*. *PreArcs* are all arcs that point from a place to a transition and *PostArcs* are all arcs that point from a transition to a place. All places, transitions and arcs have an ID and a set of attributes as it can be seen in Figure 4.2.

Finally the *persistence* component is responsible for storing and restoring Petri nets and rules as PNML (Petri Net Markup Language [Sti05], [BCVH$^+$03]) files. PNML is an XML-based syntax for high-level Petri nets with the aim to enable Petri net tools to exchange Petri net models. A secondary model that uses an object for each Transition, Place and Arc and for each of their attributes is used by a *Converter* to write ReConNets Petri net model into a PNML file.

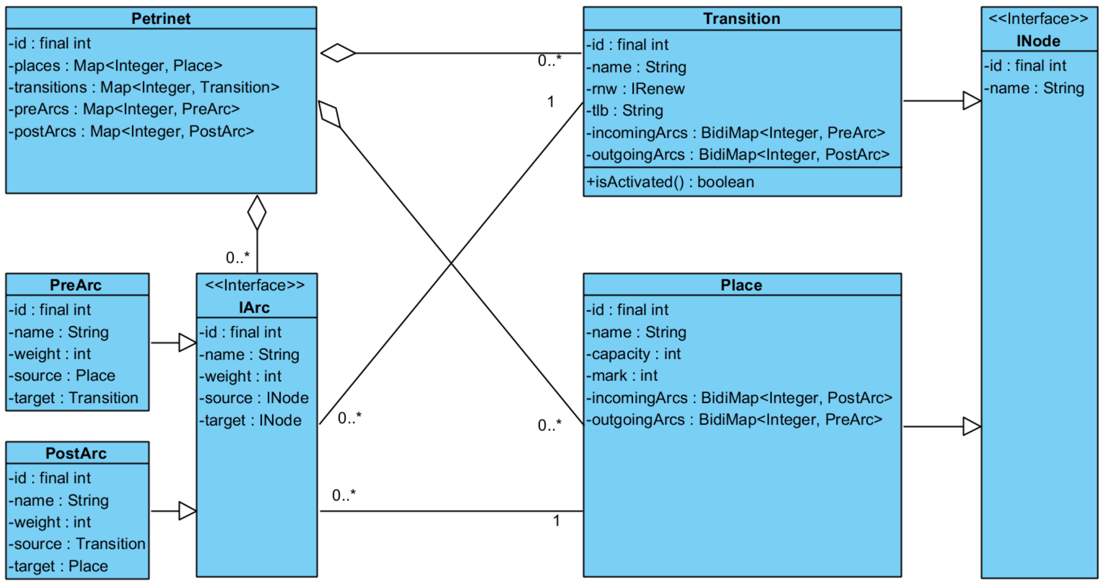


Figure 4.2.: Structure of ReConNets Petri net model.

4.1. HPN Structure

To be able to simulate hierarchical Petri nets in ReConNet several additions to ReConNet are made. Mainly the Petri net model is extended to accommodate transitions that substitute subnets.

Model changes: ReConNets model, presented in Figure 4.2, is only experiencing minor changes. The only changes to the model are made to the *Transition* class, it gains an additional attribute *subnetID* that contains the ID of the Petri net it substitutes or a negative value if it is a normal transition. Since substitution transitions cannot be activated or fired any transition that has a positive *subnetID* attribute value always return false on its *isActivated()* function that signalizes if the transition is activated and thus it never fires.

Subnet data structure: Like the data for all Petri nets and rules, the *engine* component stores the information about all subnets in an appropriate data structure. For this it uses *SubnetAttribute* objects. For each subnet exists exactly one *SubnetAttribute* that holds:

- the ID of the subnet,
- a reference to the substitution transition that is substituting the subnet,

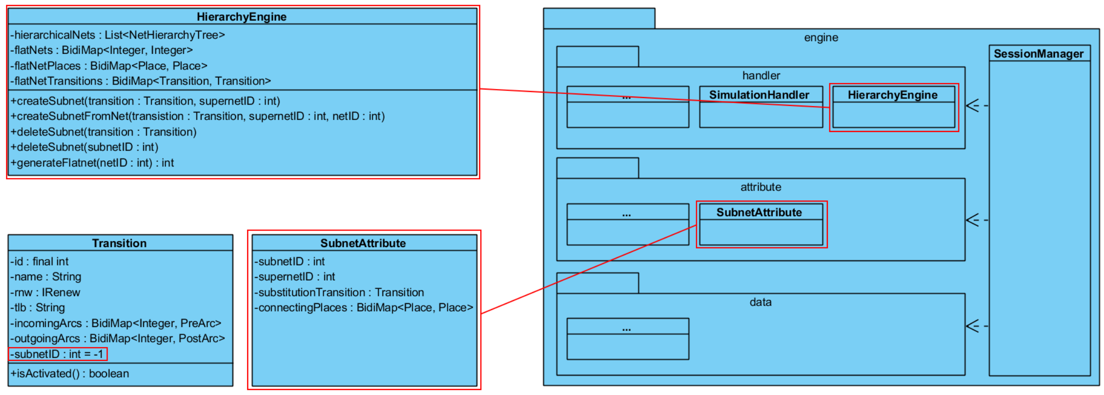

HierarchyEngine
-hierarchicalNets : List<NetHierarchyTree>
-flatNets : BidiMap<Integer, Integer>
-flatNetPlaces : BidiMap<Place, Place>
-flatNetTransitions : BidiMap<Transition, Transition>
+createSubnet(transition : Transition, supernetID : int)
+createSubnetFromNet(transistion : Transition, supernetID : int, netID : int)
+deleteSubnet(transition : Transition)
+deleteSubnet(subnetID : int)
+generateFlatnet(netID : int) : int
Transition
-id : final int
-name : String
-rnw : IRenew
-tlb : String
-incomingArcs : BidiMap<Integer, PreArc>
-outgoingArcs : BidiMap<Integer, PostArc>
-subnetID : int = -1
+isActivated() : boolean
SubnetAttribute
-subnetID : int
-supernetID : int
-substitutionTransition : Transition
-connectingPlaces : BidiMap<Place, Place>
engine
handler
...
SimulationHandler
HierarchyEngine
attribute
...
SubnetAttribute
data
...
SessionManager

Figure 4.3.: Additions to ReConNets *engine* component to accommodate hierarchy.

- the ID of the net containing this substitution transition (also known as the super net of that subnet),
- and a bidirectional mapping of all the substitution transition's connecting places in the super net to their corresponding connecting places in the subnet.

The HierarchyEngine: To manage the hierarchical nets the *HierarchyEngine* is added to the *engine* component, it maintains all *SubnetAttributes* so that their data stays accurate after changes to any hierarchical net. The *HierarchyEngine* also contains a mapping of all subnet IDs to their *SubnetAttributes* and structures all subnet IDs in *NetHierarchyTree* tree structures so that for every hierarchical net in ReConNet there exists a *NetHierarchyTree* that resembles the hierarchical net and holds all (sub-)net IDs that belong to that hierarchical net with their child-parent relations. Since all Petri (sub-)nets are stored and administrated independently via their ID by the *SessionManager*, these trees of IDs also define the hierarchy of all hierarchical nets. Figure 4.4 displays an exemplary *NetHierarchyTree* with mappings of its subnet IDs to their *SubnetAttributes*.

For the administration of the hierarchical nets the *HierarchyEngine* also provides the necessary functions to add and remove subnets. Added subnets can either be empty or a replica of a Petri net that has already been loaded into ReConNet. If the added subnet is empty, connecting places that correspond to the substitution transitions connecting places are automatically created within the subnet. If the new subnet is a replica, then for each of the substitution transitions connecting places either one place of the subnet has to be chosen as the corresponding connecting place or a new connecting place is created in the subnet.

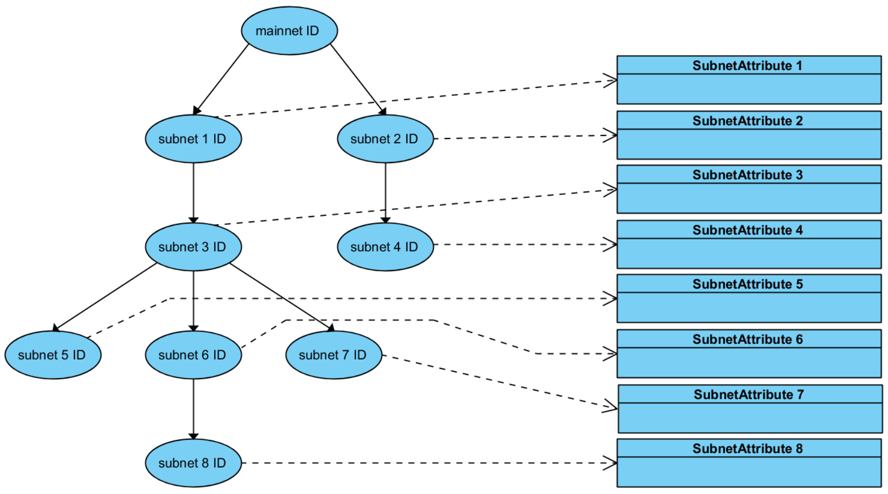


Figure 4.4.: (*NetHierarchyTree*) Tree structure for net hierarchy with the associated *SubnetAttributes*.

4.2. Persistence

To store Petri net models, PNML files are used. The *persistence* components *Converter* first creates a *Pnml* object that than is written into a PNML file.

PNML in ReConNet

A *Pnml* object contains

- a *nodeSize* for layout purposes,
- a *type* String that indicates if the PNML is for a rule (*type* = „*rule*“) or a normal Petri net (*type* = „*petrinet*“),
- and a list of *Nets* where each *Net* represents a Petri net.

The *Pnml* - *type* is mainly used as header information when restoring a Petri net or rule from PNML to chose the correct restoration procedure. In the list of *Nets*, each *Net* object represents a Petri net. It has an ID String, that correlates with the ID of the Petri net in ReConNet it represents, a *Page* that contains all the data about the Petri net and a *nettype* String. In the case the PNML is for a rule the *nettype* String contains L, K or R depending on which part of the rule the *Net* represents, if the PNML is not for a rule the *nettype* String is empty.

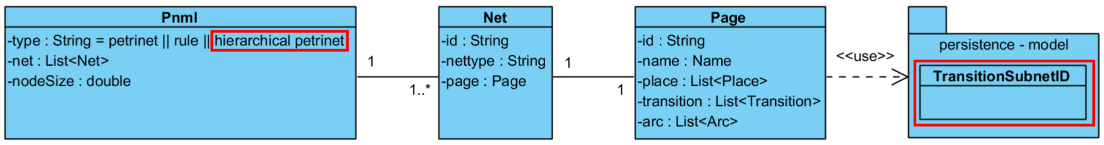


Figure 4.5.: PNML in ReConNet.

Persistence for hierarchical Petri nets

For the storage of hierarchical Petri nets a third *Pnml - type* (*type* = „*hierarchical petrinet*“) is introduced. Also when a hierarchical Petri net is converted into a *Pnml* object, the HPN's main net is the first entry in the list of *Nets*. Then all subnets follow in arbitrary order.

When restoring a hierarchical Petri net from *Pnml* first the HPN's main net is restored. Then for every transition with a subnetID in the main net, the *Net* with the corresponding ID is found in the list of *Nets* and added as subnet to that transition, this is repeated recursively for all subnets until the whole hierarchical Petri net has been restored.

4.3. Simulation

In a HRPN activated transitions on any level of the hierarchical net can fire and since the *SimulationHander* of the *engine* component is driving the simulation adjustments would have to be made there as well.

With regard to not only to the addition of hierarchy, but also the addition of local rules later, it was decided to use the flattened representations of HRPNs during the simulation. This is possible because, as Section 3.3.2 described, a HRPNs hierarchical representation and its flattened non-hierarchical representation are of equal behavior. Since ReConNet is already capable of simulating non-hierarchical nets this keeps the changes to the *SimulationHander* to a minimum.

During the design phase of a HRPN, in which the net designer develops the nets and transformation rules, true hierarchy is used and at the beginning of the simulation the flat net is acquired with the flattening process.

Then during the simulation ReConNets simulation engine switches to the flat representation of a HRPN for transition firing and transformation rule application. However for the user this remains transparent and the visual interface remains in a hierarchical view. While transitions are fired and transformations are performed on the flat net, the hierarchical net applies the changes appropriately. To achieve this the *HierarchyEngine* holds a bidirectional mapping of all places and transitions of the generated flat net to their counter parts in the HRPN. When ever tokens are moved or transformations performed on the flat net the change is mirrored to the HRPN via this mapping, so the user perceives the simulation of the hierarchical net.

4.4. Rule Application

The implementation of the local rules, described in Chapter 3, that target a specific (sub-)net of a HRPN, presents a certain challenge. Since for the simulation the flat net is used and for the application of rules one single name space for places and transitions (A_P, A_T) is needed, this name space needs to include all of the disjoint name spaces of all of the hierarchical nets (sub-)nets. This single name space is created during the flattening process. Whenever a subnet is inserted into its super net all places and transitions that are not connecting places gain a prefix to their names that is unique to the substitution transition that was replaced. The local rules meant for that subnet are adjusted to that naming scheme as well. This way the naming preserves hierarchy borders and (sub-)net identities and so the names of places and transitions are specific enough that a rule meant for only a specific (sub-)net can be limited to the correct part of the flat net. The concrete steps and name changes that are taken during the flattening process are described in Section 4.5.

To apply a rule as local rule to a (sub-)net, a (sub-)net can choose any rule that has been loaded into ReConNet. The chosen rule is then copied and associated with the (sub-)net. The *HierarchyEngine* manages local rules by keeping a mapping of (sub-)net IDs to rule ID sets that have been chosen as local rules.

4.5. HRPN Flattening in ReConNet

With the additions and changes of the previous sections in place the only part remaining that is needed for the simulation of HRPN is the flattening process. As Section 3.2.2 explained the flattening process needs to integrate all subnets into the main net and unify the disjoint name spaces of all (sub-)nets of the HRPN into one.

Naming convention

For the generation of the flat nets name space a distinct naming convention is utilized. As a first naming step, during the design phase of a hierarchical Petri net in ReConNet to every substitution transition the name prefix „ST_“ and to every connecting place the name prefix „CP_“ is added. These name prefix are are always in the front and are unavailable for other places and transitions. They separate the name spaces of connecting places and substitution transitions from other places and transitions to realize the concepted naming function for places $p_{name}P \to A_P^N$ with $p_{name}(cP) \subseteq A_{cP}$ and $p_{name}(P \backslash cP) \subseteq A_P^N \backslash A_{cP}$ and transitions $t_{name}T \to A_T^N$ with $t_{name}(sT) \subseteq A_{sT}$ and $t_{name}(T \backslash sT) \subseteq A_T^N \backslash A_{sT}$.

Secondly every time a substitution transition is flattened all places and transitions from its subnet gain a prefix to their name, in form of the name of the substitution transition. This way the subnet identity of places and transitions is retained in the flat net so that the application of a local rule meant for that subnet can be confined to that part of the flat net. Excluded from the addition of a prefix are the connecting places that form a connection set with the substitution transitions connecting places, because they are merged with their set partner places. The names in a local rule of a subnet need to be adjusted to this naming as well, so whenever a naming prefix is added to the places and transitions of a subnet they are also added to the places and transitions of every local rule that belongs to that subnet. This addition of subnet name prefixes realizes the disjoint union of name spaces during the flattening process required by the concepted model presented in Chapter 3.

Since the designer retains a hierarchical view only the substitution transition prefix „ST_“ and the connecting place prefix „CP_“ are visible, all other prefixes are invisible from the designers view to uphold transparency.

Figure 4.6 illustrates an exemplary HRPN with one local rule and the flat net of the HRPN created once with and once without subnet name prefixes. The local rule should only be applicable to the place labeled $p1$ in the subnet *subnet2*. Without the subnet naming prefixes a second occurrence for the rule arises in the flat net, which would result in different behaviour of the flat net compared to the hierarchical net which must be prevented to suite the model. The flat net with subnet name prefixes averts this issue.

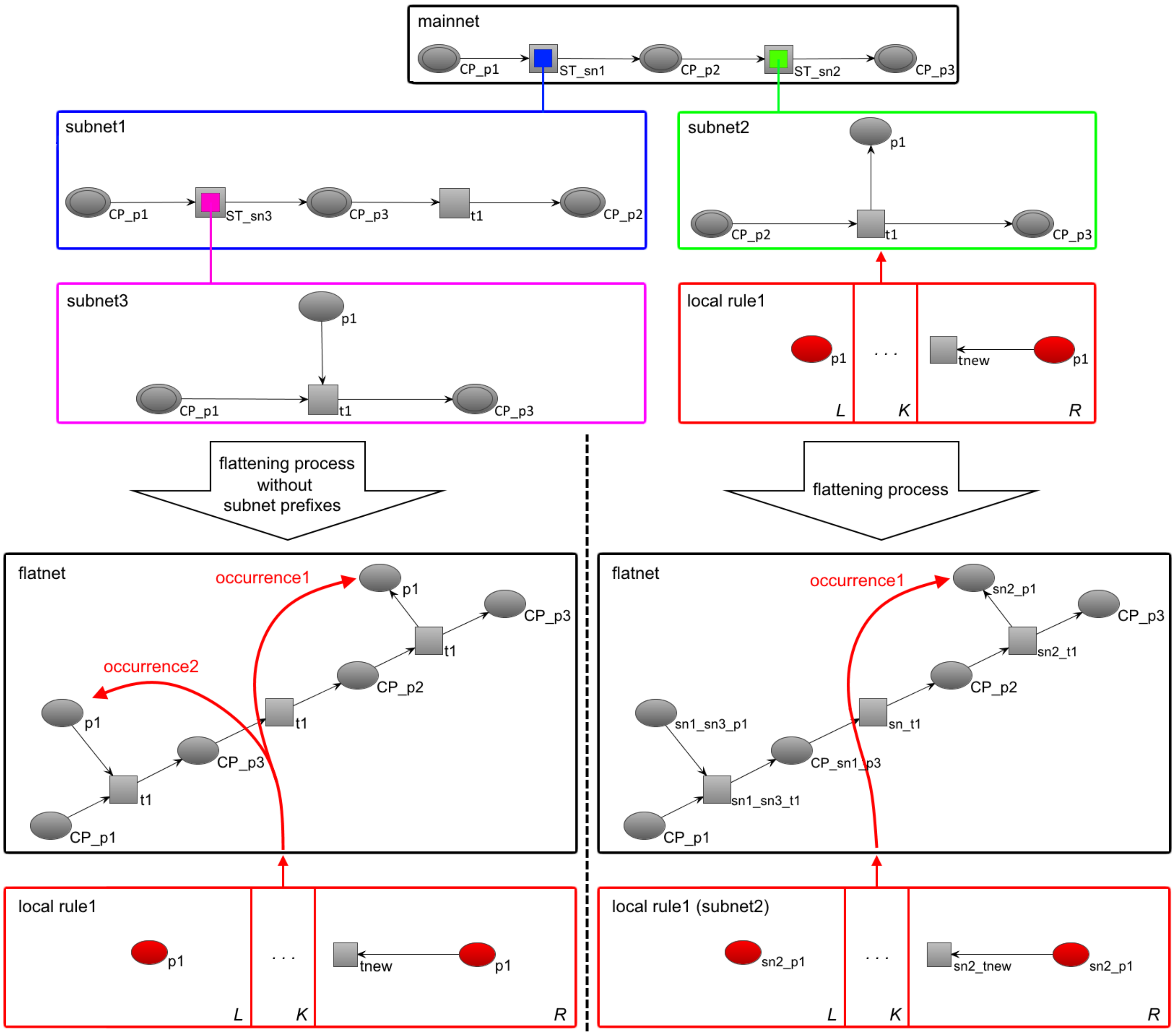


Figure 4.6.: HRPN flattening with name prefixes.

Flattening Algorithm

The flattening process that is described in Chapter 3 transforms the hierarchical net that it is applied to into the flat net. However the hierarchical representation of the HRPN is still needed after the generation of the flat net, so the first step for the generation of the flat net in ReConNet is to copy the HRPN that is to be flattened.

The flattening process is formally described as:

Definition 4.1 (Flattening Process) *The flattening is defined for an hierarchical net $HN = (RN, A, SR)$ given by a reconfigurable net $RN = (N, \mathcal{R}^N)$, a name space $A = (A_P^N, A_T^N)$ and a set of substitution rules SR as given in Def. 3.2 recursively by*

1. $flat((N, \mathcal{R}), A, SR) = (N, \mathcal{R})$ *if* $sT = \emptyset$
2. $flat(RN, A, SR) = flat(FLAT((N, \overline{\mathcal{R}}), SR), \overline{A}, \overline{SR})$ *with*
   - $\overline{A} = \biguplus_{st \in sT} (\widehat{A_{st}} \setminus A_{cP}) \uplus A_{cP}$
   - $\overline{\mathcal{R}} = \biguplus_{st \in sT} \widehat{\mathcal{R}_{st}}$
   - $\overline{SR} = \biguplus_{st \in sT} \widehat{SR_{st}}$

with $FLAT((N, \overline{\mathcal{R}}), SR) = (\overline{N}, \overline{\mathcal{R}})$ *by applying each substitution s once, $s \in S$ from Definition 3.3.*

Since Chapter 3 explains that all the transformation sequences of $HN \xRightarrow{\sum_{s \in S} s} F$ are equivalent and well-defined, thus any sequence of substitution transition flattening is fine. So for the flattening in ReConNet a bottom-up approach is chosen. It only flattens substitution transitions with subnets that contain no substitution transitions themselves. This approach avoids the creation of substitution rules that create substitution transitions.

Algorithm 1 describes the main steps that are taken in ReConNet to generate the flat net.

As can be seen in line 2 of Algorithm 1, first a copy of the hierarchical net is created. This $netcopy$ of the HRPN is then flattened. A HRPN is organized like a tree structure where every node is a (sub-)net and all leaf nodes are non-hierarchical subnets that thus do not contain substitution transitions or subnets of their own. This structural property is utilized by the recursive procedure $flattenNet$, that realizes the net flattening. The recursive procedure starts at the main net, the root of the net hierarchy tree, and tries to flatten all its subnets first, if one of its subnets has a subnet of its own it trys to flatten that one recursively first before returning to its parent net and so the procedure recursively moves down the net hierarchy tree (*lines 8-10*).

Algorithm 1 Flat net generation:

```
1: function GENERATEFLATNET($mainnet$)
2:     $netcopy$ = DEEPCOPYNET($mainnet$)
3:     FLATTENNET($netcopy$)
4:     $flatnet = netcopy$
5:     return $flatnet$
6: end function

7: procedure FLATTENNET($net$)
8:     for each subnet $s$ of $net$ do
9:         FLATTENNET($s$)
10:    end for
11:    for each subnet $s$ of $net$ do
12:        for each local rule $r$ of $s$ do
13:            ADDSUBNETNAMEPREFIXESTORULE($r$, SubnetAttribute of $s$)
14:            add $r$ to the local rules of $net$
15:        end for
16:        $sr$ = CREATESUBSTITUTIONRULE(SubnetAttribute of $s$)
17:        transform $net$ by applying $sr$
18:    end for
19: end procedure
```

When reaching a leaf node (Definition 4.1 case 1.) the flattening procedure does nothing, because all leaf nodes are non-hierarchical nets already. After returning from the last child of a (sub-)net net whose child nodes are all exclusively leaf nodes the procedure flattens all substitution transitions of net (*lines 11-18*) (Definition 4.1 case 2.). For this all of nets children's local rules are modified with subnet name prefixes and added to nets own set of local rules (*lines 12-15*). Afterwards a substitution rule is created for each of nets substitution transition and then applied (*lines 16-17*). This realizes Definition 4.1s union of name spaces and rules, a union of substitution rules in unnecessary since due to their lack of substitution transitions the subnets do not contain substitution rules. As a result net is turned into a leaf node itself which enables its parent node to be flattened by the procedure.

Figure 4.7 displays the (sub-)net-wise flattening of an exemplary hierarchical net tree using this algorithm.

Substitution Rules and Subnet Prefixes

During the flattening process the subnet prefixes that were first described in Section 4.5 are constructed. They are introduced into to the net through the substitution rules. Figure 4.8

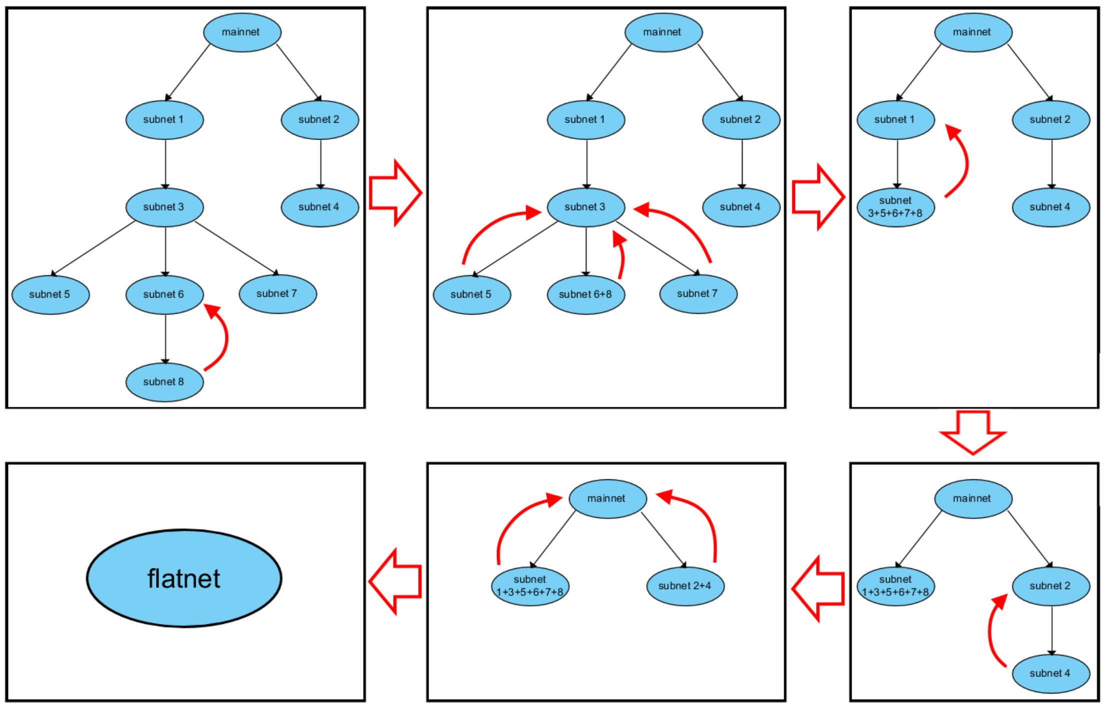


Figure 4.7.: Tree view of a HRPN flattened in ReConNet.

shows the substitution rule for the substitution transition ST_sn2 of the HRPN displayed in Figure 4.10.

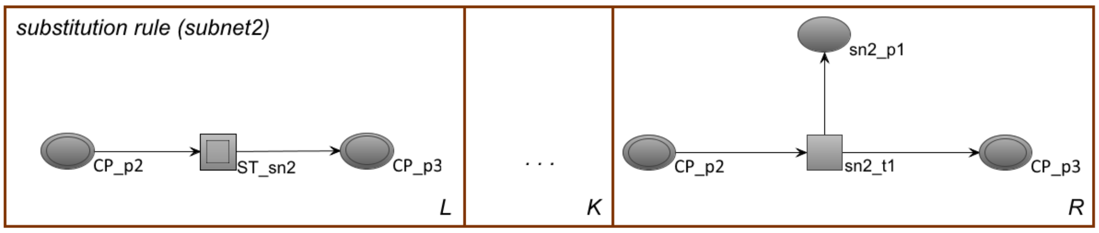


Figure 4.8.: Substitution rule with subnet name prefixes.

Substitution transition ST_sn2 substitutes the subnet $subnet2$, so for the creation of the substitution rule of Figure 4.8 the left-hand side L contains the substitution transition and its surrounding net, i.e. its connecting places. The right-hand side R contains the subnet $subnet2$ but to each place and transition of R the name of the substitution transitions without the ST_ prefix is prepended. Excluded from this are the connecting places CP_p2 and CP_p3

since they are part of ST_sn2s connection sets. So in R the place and transition $p1$ and $t1$ are named $sn2\_p1$ and $sn2\_t1$ respectively.

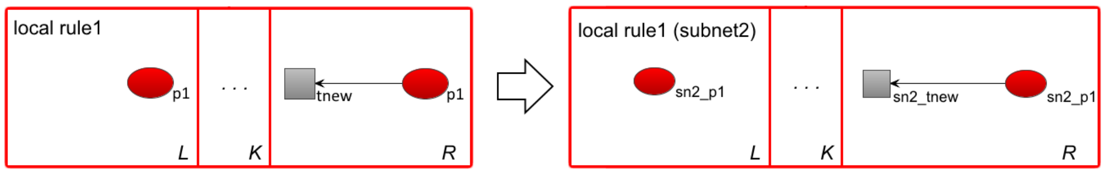


Figure 4.9.: Local rule before and after renaming with subnet name prefixes.

In a similar way subnet prefixes are also added to local rules. When for example Figure 4.10s substitution transition ST_sn2 is being flattened, all of $subnet2$s local rules are added to the net that contains ST_sn2 (here: $mainnet$). And in all L,K and R nets of local rules that are transferred this way, the names of places and transitions are prependend with the substitution transitions name without the ST_ prefix. The exception for connecting places that correlate to the substitution transitions connection sets applies here as well. So the places named $p1$ are renamed to $sn2\_p1$ and the transition $tnew$ is renamed to $sn2\_tnew$.

In Figure 4.9 $local\ rule1$ can be seen before and after this renaming.

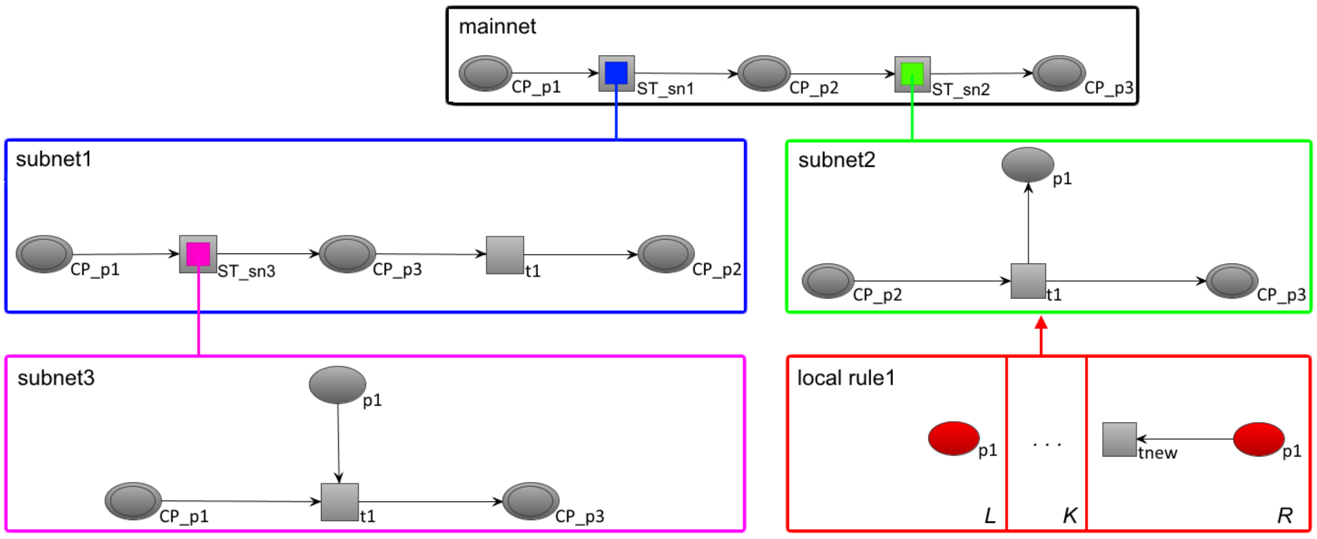


Figure 4.10.: A HRPN example.

5. Evaluation of HRPN in ReConNet

Since other Petri net models struggle with the modeling of dynamic reconfigurable hardware like FPGAs or flexible manufacturing systems and reconfigurable Petri nets are meant solve this issue, it seems appropriate to evaluate the added hierarchical reconfigurable Petri net modeling capabilities of ReConNet by using such a system as an example. So for the evaluation of ReConNet this chapter first presents an exemplary flexible manufacturing system (FMS) and then utilizes ReConNet to model said system. Afterwards the modeling procedure as well as the modeled HRPN are evaluated to see if the modeling of HRPN in ReConNet hold advantages over similar modeling procedures and tools.

5.1. The FMS Evaluation Example

In Section 2.4 the basic concepts of FMS have already been explained. A FMS is a group of computer numerically controlled machines (CNC) that are central controlled and connected through an automated transport system. The following FMS evaluation example seeks to be as simple as possible while being complex enough to appropriately demonstrate the modeling a real FMS with HRPN.

The FMS evaluation example can produce four kinds of completed parts: toy bricks, gears, clocks and toy cars. For the production the system takes in six different starting work parts: green, blue and red plastic granulate, aluminum and steel sheets and electric engines. Starting work parts and completed parts enter and leave the FMS through a loading and unloading station. Part of the FMS are four CNC machines. One of the CNC machines is an injection molder, two are laser cutters and one is an assembly machine. The CNC machines are connected to a conveyor transport system with a loop-layout so that parts can be transported from any CNC machine to any CNC machine. An overview of the FMS is given in Figure 5.1.

CNC1: CNC1 is the injection molder, the injection molder takes in green, blue and red plastic granulate, melts the mixed granulate down and injects the molten plastic into molds. By mixing differently colored granulate a wide spectrum of colored plastic can be created. Also, independent from the used plastic mixture, different molds can be used to create different

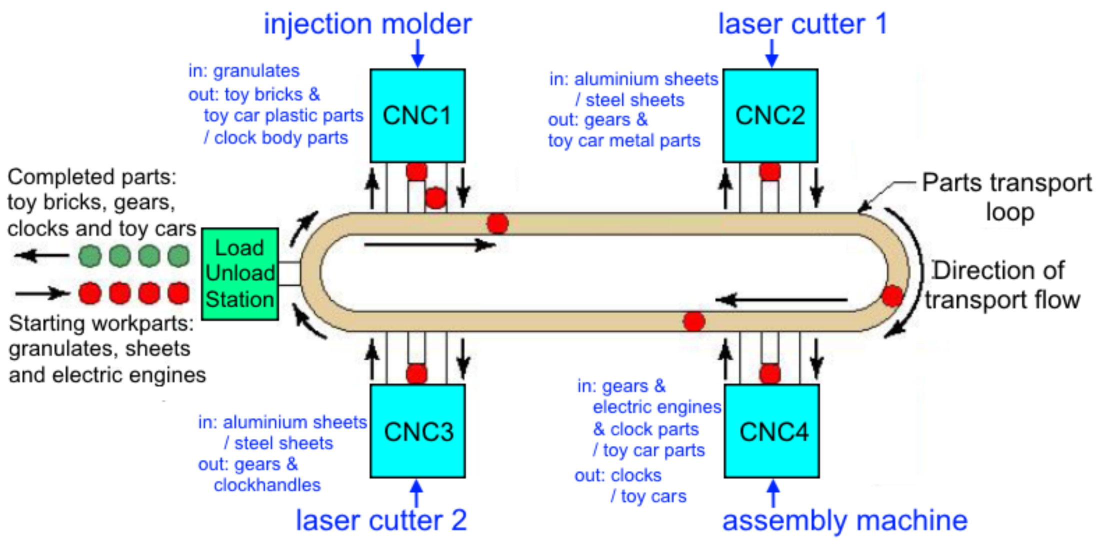


Figure 5.1.: Overview of the FMS evaluation example.

plastic parts. For this FMS the injection molder makes either toy bricks and parts for toy cars or toy bricks and clock body parts. All parts are created in one of two possible colors.

CNC2 & CNC3: CNC2 and CNC3 are laser cutters (laser cutter 1 and laser cutter 2, respectively). The laser cutters take in either aluminum or steel sheets, one kind of material is used until it runs out. A computer controlled laser then cuts a sheet into parts. For this FMS, laser cutter 1 (CNC2) cuts toy car metal parts and laser cutter 2 (CNC3) cuts clock handles. Both laser cutters use spare room on the metal sheets to cut out gears. The gears are then galvanized and finally all parts are polished before leaving the CNC machine. CNC2 can also add or remove a galvanization step to the production of toy car metal parts on the fly.

CNC4: CNC4 is an assembly machine. It can assemble one item at a time which can be either a clock build from clock body parts, clock handles, gears and an electric engine or a toy car build from toy car plastic and metal parts, gears and an electric engine. It can only switch from the assembly of one to the other between completed assembly processes so that no wrong parts reside within the machine.

5.2. The HRPN of the FMS

For the HRPN model of the FMS evaluation example a main net with four subnets is created. Each subnet models the behavior of one CNC machine while the main net models the behavior of the transport system and the loading and unloading station. The centralized controlling system is represented by local rules that are added to each CNC subnet and the limited number of edges in the main net.

The following sections presents the most important parts of the created HRPN, the complete HRPN with close ups of all nets, rules, the flat net and subnet templates can be found in Appendix A.

Main Net

The main net is displayed in Figure 5.2. The transitions with only one edge on the far left and right side of the net add starting work parts to and remove completed parts from the system. These transitions function as the FMS's loading and unloading station. All places of the main net are part of the FMS's transport system, they connect the loading and unloading station and all CNC machines. The transitions ST_CNC1, ST_CNC2, ST_CNC3 and ST_CNC4 are substitution transitions, they house the subnets for the machines CNC1 (injection molder), CNC2 (laser cutter 1), CNC3 (laser cutter 2) and CNC4 (assembly machine).

In essence: Tokens representing starting work parts enter the system trough the transitions on the left side, the CNC subnets take these tokens in and produce tokens representing completed parts that exit the system through the transitions on the right side. Furthermore there are no (local) rules allotted to the main net itself.

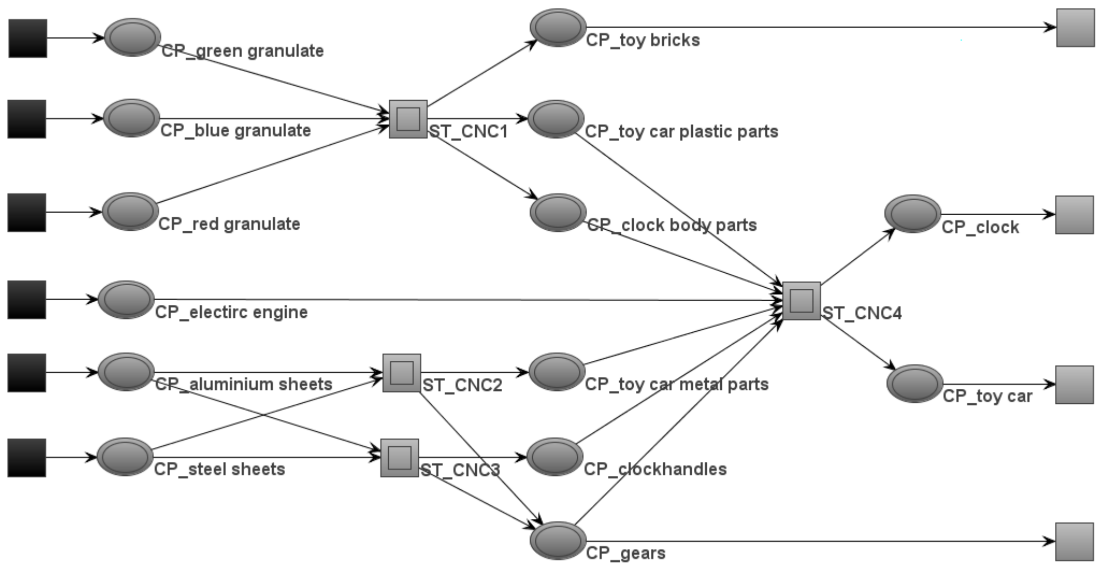
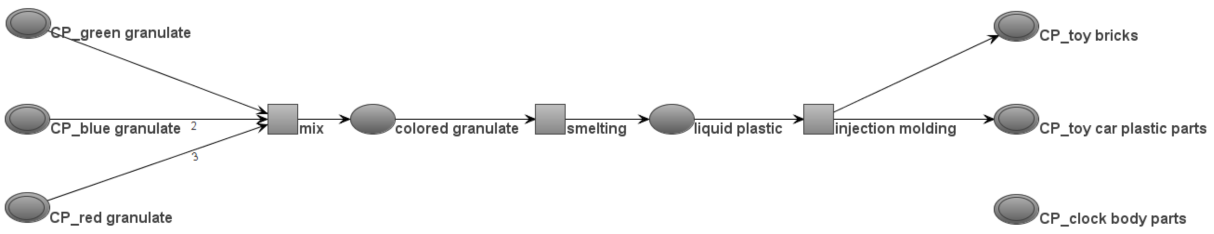


Figure 5.2.: Main net of the HRPN.

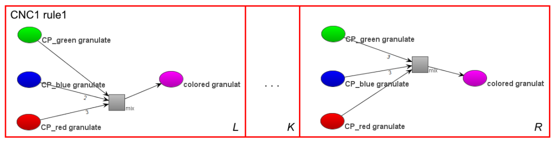
CNC1 - Injection molder

In Figure 5.3 (a) the subnet CNC1 is displayed. It models the behavior of the injection molder. The places $CP\_green\ granulate$, $CP\_blue\ granulate$ and $CP\_red\ granulate$ are connecting place through which granulate tokens enter the subnet. As a first step differently colored granulates are mixed in a predefined ratio. The mixed colored granulate is then smelted and injected into molds. The tokens representing the molded parts exit the subnet through the connecting places $CP\_toy\ bricks$, $CP\_toy\ car\ plastic\ parts$ and $CP\_clock\ body\ parts$.

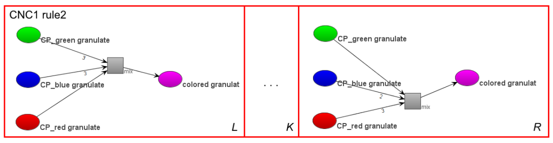
In its starting configuration the injection molder produces toy bricks and toy car plastic parts from plastic that uses one part green, two parts blue and three parts red granulate. The FMS can switch the mixing ratio from (1 green / 2 blue / 3 red) to (3 green / 3 blue / 1 red). The local rules $CNC1\ rule\ 1$ and $CNC1\ rule\ 2$, displayed in Figure 5.3 (b) and Figure 5.3 (c), model the switch between the two compositions by changing the weights on the edges leading into the transition mix. Rule 1 switches the ratio from (1 green / 2 blue / 3 red) to (3 green / 3 blue / 1 red) and rule 2 can reverses this switch. The local rules $CNC1\ rule\ 3$ and $CNC1\ rule\ 4$, displayed in Figure 5.3 (d) and Figure 5.3 (e) model the switch between the production of toy car plastic parts and clock body parts by changing the edges coming from the $injection\ molding$ transition. Rule 3 switches from toy car plastic parts to clock body parts and rule 4 can reverse this change.



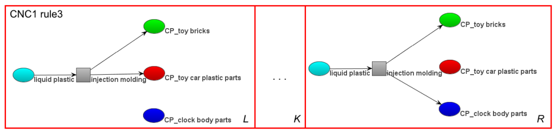
(a) Subnet CNC1



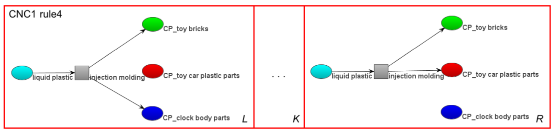
(b) CNC1 rule 1: granulate composition switch.



(c) CNC1 rule 2: granulate composition reversal.



(d) CNC1 rule 3: mold switch.



(e) CNC1 rule 4: mold reversal.

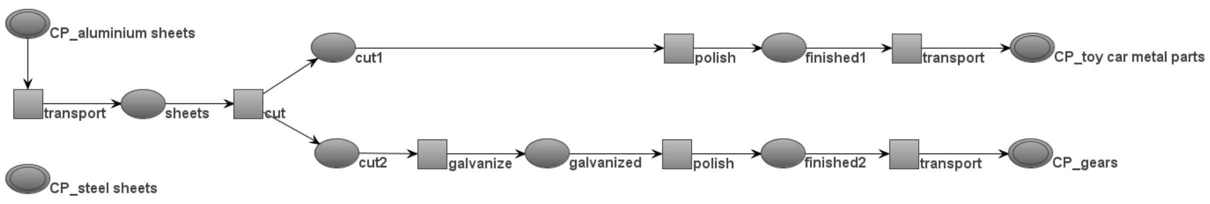
Figure 5.3.: Subnet CNC1 with its local rules.

CNC2 & CNC3 - Laser Cutter 1 & 2

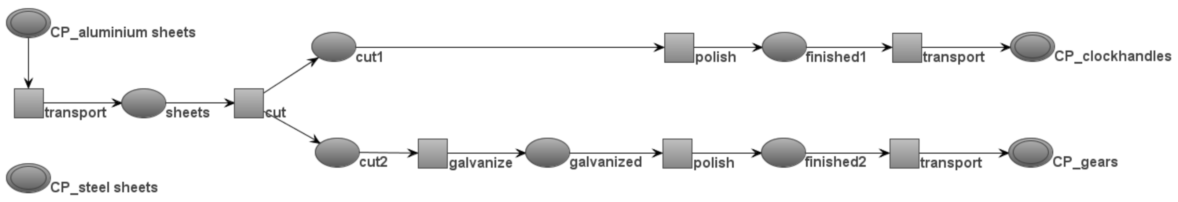
The subnets CNC2 and CNC3 are displayed in Figure 5.4 and they are both created from the same template net. On the left side of the net metal sheet tokens enter the subnet through the places $CP\_aluminum\ sheets$ and $CP\_steel\ sheets$. The sheets are then cut into gears and metal parts for toy cars (CNC2) or clock handles (CNC3). Metal parts are polished and transported out while gears are galvanized, polished and then transported out of the subnet through the connecting places CP_gears, $CP\_toy\ car\ metal\ parts$ and $CP\_clockhandles$.

The local rules displayed in Figure 5.5 model the switch between the two sheets variants by changing an edge between the current sheet material place and the the transition $transport$ to the alternative sheet material place. Each rule is fitted with a NAC with a single token on the place of current sheet material, so that the rule is only applicable when no more tokens of the current material are present.

As Section 5.1 described CNC2 can add and remove a galvanization step for toy car metal parts. The rule displayed in Figure 5.6 (a) models the addition and the rule in Figure 5.6 (b) models the removal of the galvanization step. $CNC2\ rule\ 1$ adds a place and transition after the cutting step and $CNC2\ rule\ 2$ removes them.

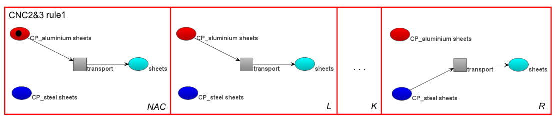


(a) Subnet CNC2

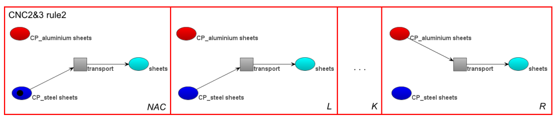


(b) Subnet CNC3

Figure 5.4.: Subnet CNC2 and CNC3.

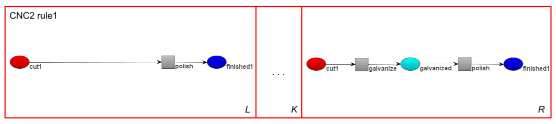


(a) CNC2&3 rule 1: sheet material switch.

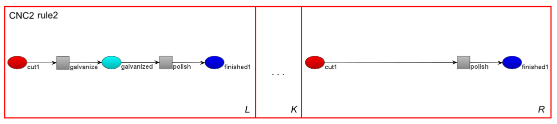


(b) CNC2&3 rule 2: sheet material reversal.

Figure 5.5.: Local rules for CNC2 and CNC3.



(a) CNC2 rule 1: add galvanization step.

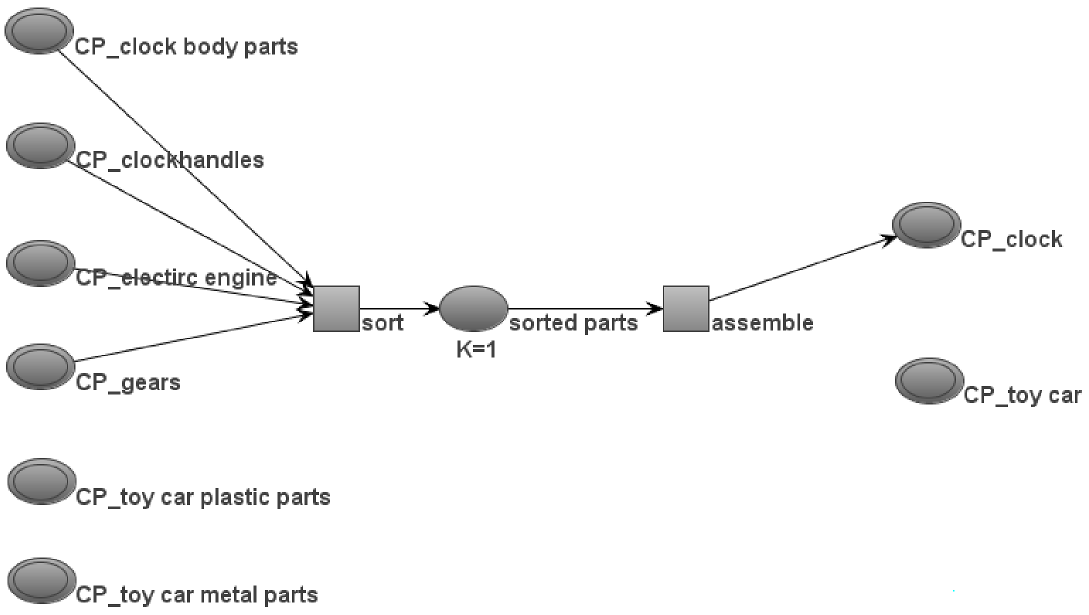


(b) CNC2 rule 2: remove galvanization step.

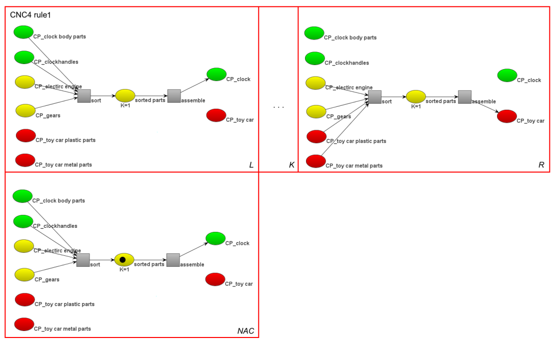
Figure 5.6.: Local rules for CNC2 only.

CNC4 - Assembly Machine

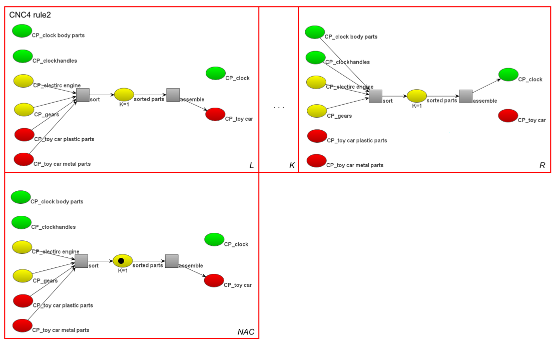
Figure 5.7 displays the subnet CNC4 with its two local rules. The subnet takes tokens representing clock or toy car parts from its connecting places in the left via its *sort* transitions and moves them through its *assemble* transition to the connecting places CP_clock and CP_toy car. Since the assembly machine can only assemble one item at a time the place *sorted parts* has a capacity of one.



(a) Subnet CNC4



(b) CNC4 rule 1: switch to toy car assembly.



(c) CNC4 rule 2: switch to clock assembly.

Figure 5.7.: Subnet CNC4 with its local rules.

The switch between the assembly of clocks and toy cars is model through CNC4s local rules $CNC4\ rule$ 1 and $CNC4\ rule$ 2. The two rules change edges to and from the subnets connecting places so that the appropriate part tokens are taken in by the $sort$ transition and the corresponding finished product token is created by the $assemble$ transition. Each rule has a NAC with one token on the place $sorted\ parts$ so that it is assured that the rule can only be applied when the CNC machine is empty.

5.3. Simulation

From the HRPN presented in Section 5.2, ReConNet generates the flat net displayed in Figure 5.8 and a set of twelve rules. The nets that replaced the former substitution transitions are marked in blue. The generated net and rules are used by ReConNet to simulate the HRPN.

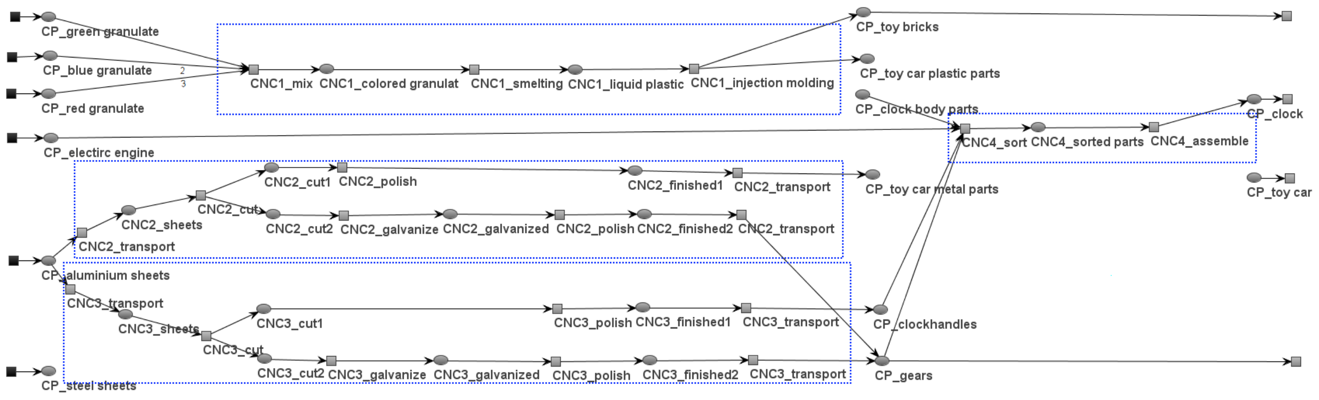


Figure 5.8.: FMS evaluation example: flat net. Former subnets are marked in blue.

As can be seen in the flat net, in its starting configuration the FMS net can neither produce clocks nor toy cars. This is because the injection molder is set to produce toy car plastic parts, so no clock body parts are being created, and the assembly machine is set to assemble clocks and thus no toy cars are being assembled. Only through reconfiguration can the model reach configurations in which these two items, represented by the places CP_clock and $CP\_toy\ car$, can be created.

To test simulate the system a fix number of resource tokens are added to the system. 300 tokens are placed onto each of the granulate places $CP\_green\ granulate$, $CP\_blue\ granulate$ and $CP\_red\ granulate$ and 100 tokens are placed on each of the places for metal sheets and electric engines ($CP\_aluminium\ sheets$, $CP\_steel\ sheets$ and $CP\_electric\ engine$). In order to avoid that further resources are added to or removed from the system the transitions representing the load and unload station are being removed. This results in the altered main net presented in Figure 5.9.

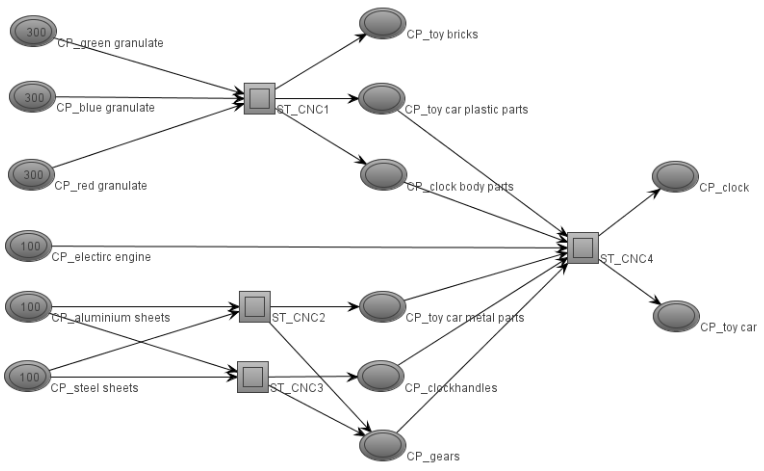


Figure 5.9.: FMS simulation example.

For the simulation test the system runs until no further transition firing can occur, at which point only transformation is still possible. Figure 5.10 illustrates the main net after about one-half of the simulation, at which point during the last step the last token from the place $CP\_aluminium\ sheets$ has just been removed. So far multiple transformations within the subnets CNC1, CNC2 and CNC4 could be observed. Multiple tokens on the outgoing connecting places of ST_CNC1 and ST_CNC4 attest to the production switch of the two subsystems. CNC3 and the left side of CNC2 on the other hand have not experienced any transformations, also no tokens have been removed from the place $CP\_steel\ sheets$. This is consistent with the premises that the laser cutter intake of sheet material only switches once one material is unavailable.

Figure 5.10 shows the main net at the end of the simulation. At this point no more starting work parts can be processed and no more completed parts can be assembled due to lack of electric engines. During the simulation it could be observed that the laser cutter subnets CNC2 and CNC3 switched their sheet material to steel and that further transformations occurred in the other subnets CNC1 and CNC4 which is also indicated by the additional tokens on the places for completed parts and plastic parts of both kinds.

When looking at the simulation observations, all observations are backed by the FMS description in Section 5.1 and the simulation supports the assertion that the presented HRPN of the FMS works correctly. Assuming the model represents the exemplary FMS correctly it can be used to make statements about the FMS itself.

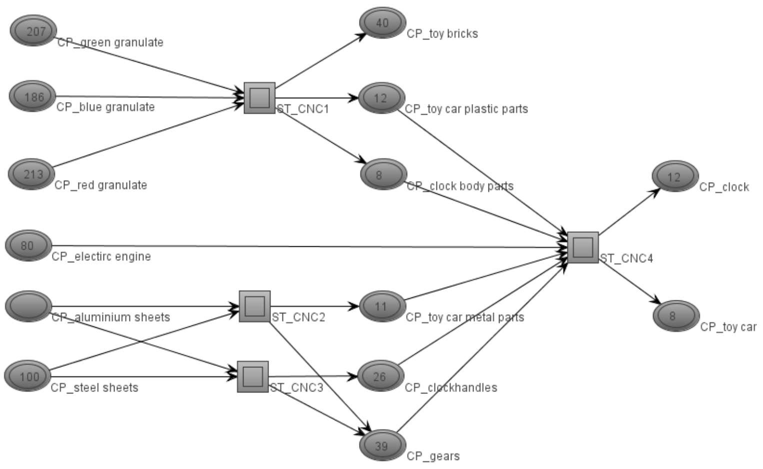


Figure 5.10.: FMS simulation example after one-half of the simulation.

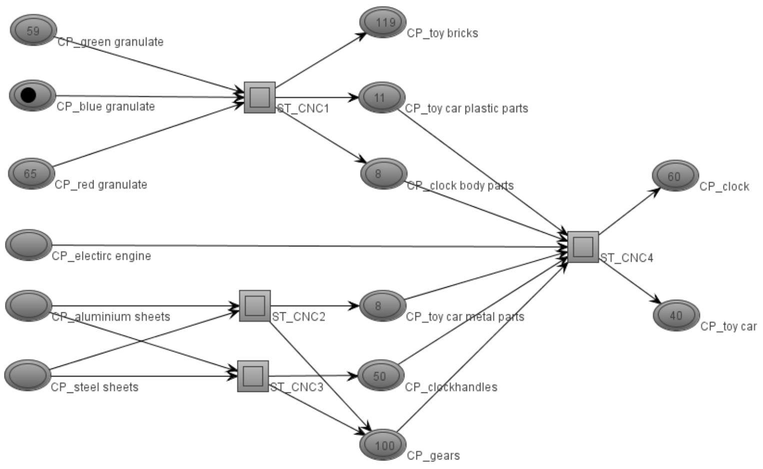


Figure 5.11.: FMS simulation example at the end of the simulation.

5.4. Evaluation

In this section the hierarchical reconfigurable Petri net model and its implementation in ReConNet are evaluated using the FMS evaluation example. First the usefulness of HRPN for modeling dynamic systems is considered in Section 5.4.1. For this the HRPN modeling approach is compared to HPN approaches without reconfiguration rules. Then Section 5.4.2 evaluates ReConNets modeling capabilities and compares them to similar tools that support hierarchical Petri nets. For the comparison the well developed and well known tool *CPN tools* is chosen as a representative.

5.4.1. Usefulness of HRPN

To evaluate the usefulness of the HRPN, this section analyzes Petri net approaches without reconfiguration to see what difficulties arise and how much effort it would be to overcome these difficulties compared to the HRPN approach.

Without the reconfiguration rules the FMS evaluation example can be modeled as a hierarchical Petri net. A simple way to do this is to create a HPN for every possible net configuration. In this simple example with only six parts of the net switching between two states this results in $2^6 = 64$ different nets. Since most net configurations are quite similar to one another after modeling the first net every of the other 63 nets can be created in only a fraction of the time. Still if one net is reworked later these change need to be propagated to all 64 nets which is a considerable amount of effort It is also to consider that the FMS evaluation example is small and of low complexity, bigger systems would result in far more nets with the number of nets rising exponentially with increasing system size and complexity of the system making this an impractical approach. Also this would disregard the fact that the system can switch between configurations on the fly which might result in crossed states unreachable by any of the single nets.

Since in the FMS evaluation example the configuration switches each only apply to a certain subnets, instead of recreating the entire net for each possible net configuration, an other simple approach is to model each subnet configuration and then add it to the main net in a parallel fashion. In the case of the FMS evaluation example this results in $2^2 = 4$ nets for CNC 1, $2^2 = 4$ nets for CNC2, 2 nets for CNC3 and 2 nets for CNC4 for a total of $4 + 4 + 2 + 2 = 12$ subnets instead of the $4 * 4 * 2 * 2 = 64$ whole nets of the first approach. Figure 5.12 displays the FMS evaluation examples main net using this approach. This approach takes significantly less effort than the first approach with 64 nets. Also if changes are made to a subnet, these changes only need to be propagated to the other configurations of the subnet. Furthermore

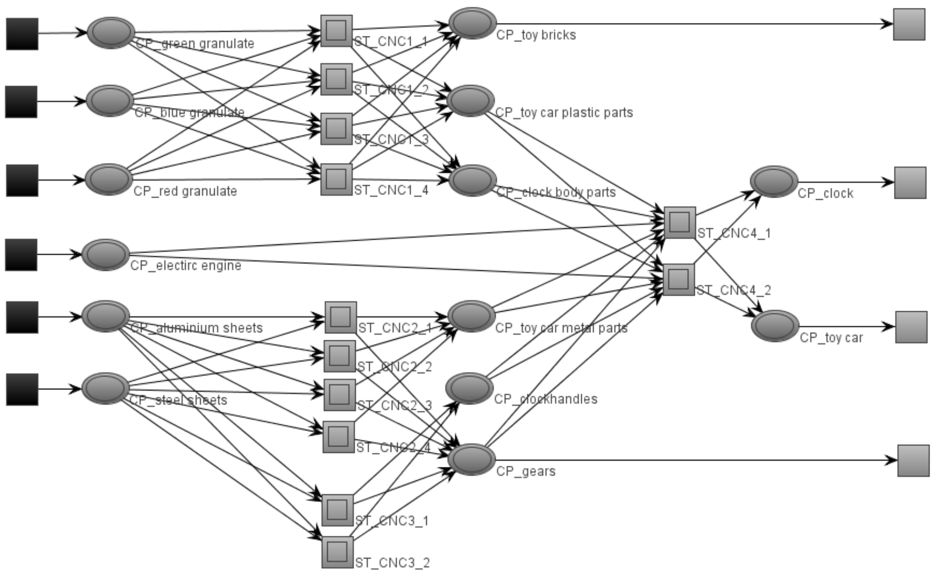


Figure 5.12.: Main net of the FMS evaluation example with separate subnets for each subnet configuration.

in contrast to the first approach the number of nets would not rise exponentially in bigger, more complex systems. With more use of hierarchy the number of duplicated subnets could possibly be reduced further.

While this approach is significantly more reasonable than to create a HPN for every possible net configuration and does account for more crossed states, since all configurations are within one HPN, it still does not account for crossed states that are created within one subsystem. An other problem is that it can also violate some system restrictions. For example, two parallel subnets can violate a restriction on CNC4 where only one item may be assembled at a time. By having a token parallel on each of the *sorted parts* places in both CNC4 subnet instances, to items would be assembled simultaneously.

A less simple approach than the first two can address the issue of crossed states. This approach adjust each switching part of the net locally. Figure 5.13 shows how this is done for CNC1.

Some system switches however have additional restrictions. In the HRPN these are realized using NACs. The switch between the assembly of clocks and toy cars in CNC4 is an example for this. Figure 5.14 shows how this is achieved using the approach of paralleling each switching part of a net locally. Since the restriction of CNC4 is that only one item is assembled at a time and that the assembly type can only be switched when the the machine is empty, the additional changes are rather limited.

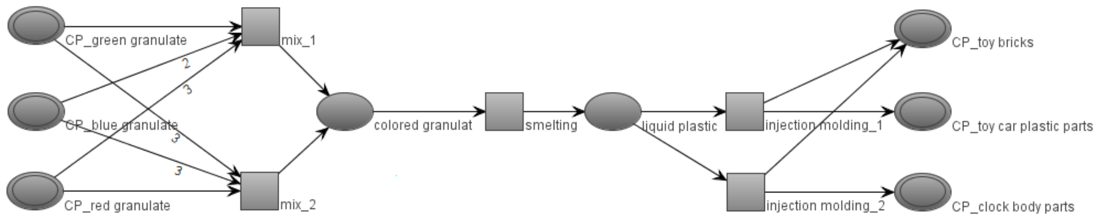


Figure 5.13.: CNC1 of the FMS evaluation example with all its configurations added locally.

CNC2 and CNC3 also have a restriction on a system switch for their sheet material. The switch from aluminum sheets to steel sheets may only occur when there are no more aluminum sheets, i.e. no tokens on the place $CP\_aluminium\ sheets$. Although this is not a complex restriction and in a HRPN modeling a NAC that accounts for this is can be done quite easily, modeling this behavior without rules and NAC is quite difficult. Other, more complex, restriction can become even harder or impossible to realize.

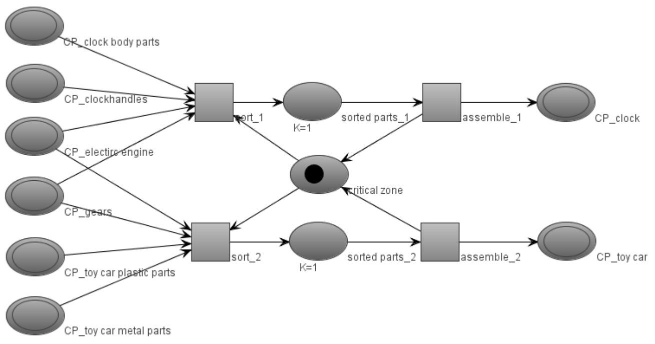


Figure 5.14.: CNC4 of the FMS evaluation example with all its configurations added locally.

In regard to systems with dynamic components it turns out the HRPN seem to hold significant advantages over non reconfigurable approaches. Although in some cases the approaches using multiple HPN or parallel subnets are feasible, the realization as a HRPN seems both easier during the first creation and during phases of revision and rework of the systems nets.

The approach using local adjustments can cope with most situations but complex restrictions or locations containing complex or multiple dynamic components would involve a serious amount of modeling work and considerations, that could be solved by using a set of rules with NAC in a HRPN. This becomes even more apparent when such a system part is reworked. Where in the HRPN only a rule needs to be added, removed or altered, the modeler using the approach with local adjustments would often need to rethink all interactions between the dynamic components.

In comparison to the non reconfigurable approaches the use of HRPN appears to be less time consuming, less complicated and more clear-cut, which also reduces the chance for error. The ability to quickly add, remove or recombine rules also gives more flexibility to the design.

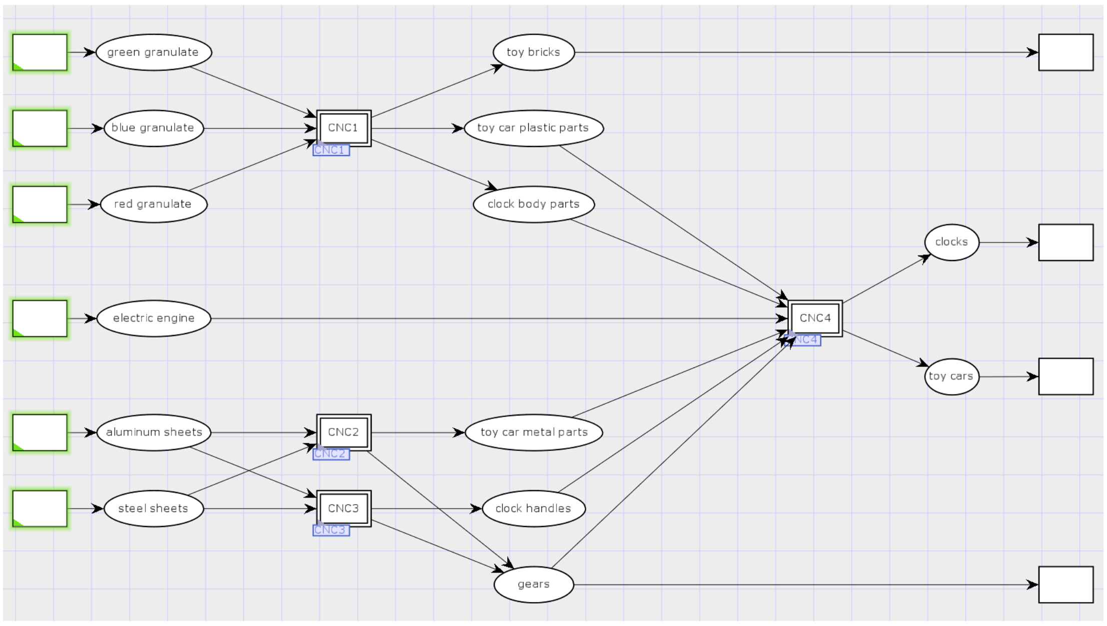
One of the strong points of hierarchical Petri nets is the reusability of subnets. The use of HRPN makes reusing subnets even more appealing since local rules can also be reused and recombined, to suit new needs, as well. An example for this in the FMS evaluation example would be adding a new laser cutter that produces clock handles, only uses aluminum sheets but adds and removes a galvanizing step to its clock handle production. Altough this configuration does not yet exist in the system this could be achieved without the need to design any new nets or rules by using the laser cutter net as template and adding the two rules responsible for the galvanizing step switch in CNC2 (*CNC2 rule1* and *CNC2 rule2*).

A disadvantage of the HRPN approach becomes apparent during validation and model checking. Compared to a HPN, checking a HRPN requires a more complex algorithm that incorporates the reconfiguration rules. However the presented HRPN can be flattened into an equivalent RPN and works like [Sch14] provide the algorithms necessary for model checking of RPN and thus for HRPN in extension.

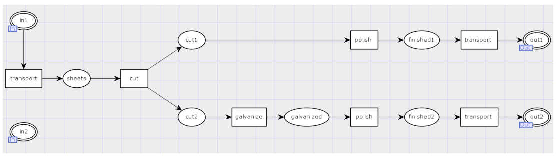
5.4.2. Hierarchy Modeling Capabilities of ReConNet

To evaluate the modeling capabilities of ReConNet we compare it to similar tools with the capability to model hierarchical Petri nets. Common tools that can be used to model HPN are for example *snoopy* [HHL+12], *CPN tools* [RWL+03] or *HiPS* [HiP17]. Since a complete survey and comparison of HPN Petri net tools is not the purpose of this section one Petri net tool is chosen as comparative example for ReConNet. A survey of different Petri net tools can be found in [TA15]. For this sections comparison the tool CPN tools is chosen, since it is well documented and readily accessible. After some general differences between CPN tools and ReConNet this section concentrates on aspects concerning the modeling of hierarchical Petri nets.

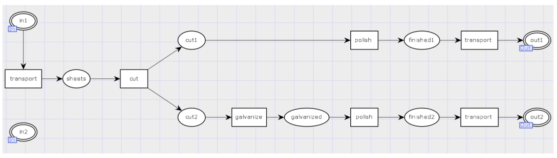
First off CPN tools uses colored Petri nets (CPN [Zim08]), which allows tokens to have a data value attached to them. Since CPN are backward compatible any normal Petri net can be modeled using a colored Petri net. ReConNet uses decorated Petri nets, this allows places to have limited capacities and transitions to have additional labels. CPN tools also provides some additional tools like syntax checking and state space analysis.



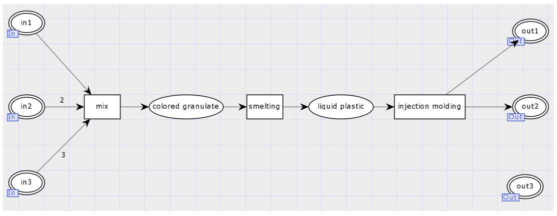
(a) FMS main net in CPN tools.



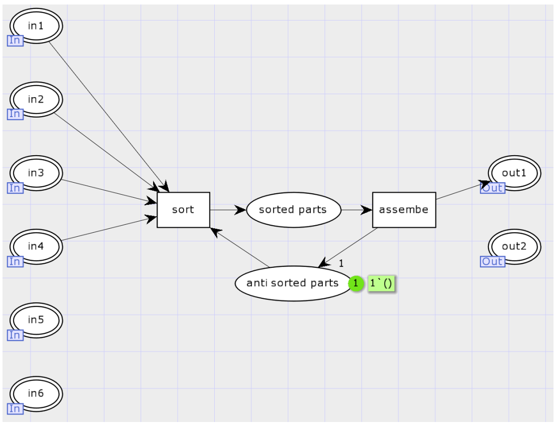
(b) FMS CNC2 net in CPN tools.



(c) FMS CNC3 net in CPN tools.



(d) FMS CNC1 net in CPN tools.



(e) FMS CNC4 net in CPN tools.

Figure 5.15.: The start configuration of the FMS evaluation example, modeled with CPN tools.

For the hierarchy evaluation the FMS evaluation example is modeled using CPN tools. Since CPN tools does not support dynamic changes the starting configuration of the FMS evaluation example described in Section 5.2 is used for the evaluation. Figure 5.15 displays the FMS evaluation example modeled with CPN tools. Since CPN tools has no decorations the place *sorted parts* in CNC4 that has a capacity of one must me modeled differently. The limited capacity is achieved by using an additional place parallel to the place *sorted parts* but with inverted arc directions and one initial token.

ReConNet and CPN tools use the same hierarchical Petri net concept. They both use substitution transitions with connecting places organized in sets of two, one place in the subnet and one place in the net with the subnets substitution transition, also known as the its super net. In CPN tools the connecting place in the subnet is called *port place* and its partner place in the super net is called *socket place*. Any place can be declared a port place and all places connected to a substitution transition are considered socket places. If a port place is in a subnet it can be assigned to one of the socket places of its substitution transition, thus building a connection set. A socket place and a port place do not have to share the same name. In ReConNet port places are automatically created. Also ReConNet has a stricter naming convention due to its reconfiguration rules and so port and socket places have to share the same name. While ReConNet can create a subnet from an existing net, picking or creating port places in the process, CPN tools has to create an empty net, copy the existing net into the empty net and then assign port places to socket places.

Both of the tools can simulate the Petri net's transition firing. In both tools this simulation behaves generally the same with the exception that CPN tools can rewind the simulation while ReConNet has to reload the net. Of course ReConNet also has the ability to simulate dynamic changes which CPN tools lacks.

Modeling the initial configuration of the FMS evaluation example using ReConNet and CPN tools reveals that ReConNets capabilities for the modeling of hierarchical Petri nets are quite similar to comparable to tools like CPN tools. Yet ReConNet lacks some comfort functions that other tools provide, functions like defining node groups, copying nodes and node groups, snapping nodes to a grid or displaying multiple Petri nets simultaneously in different windows.

6. Future Work

For future advancements of ReConNet there are several possible starting points. In Chapter 3 this work introduced hierarchy and hierarchy dependent rules in the form of global rules, layer based rules and local rules. While local rules are now available in ReConNet the addition of global rules and layer based rules could be a next step in the future development of ReConNet. [LP18] discusses an approach using a HRPN with a labeling function with an order for subtyping labels, which allows more abstract rules. Incorporating this ordered labeling could be a step to add global rules to ReConNet.

The evaluation process of Chapter 5 revealed a couple of ReConNets weak spots. Although ReConNet is formally sound a couple of additional features could improve ReConNets modeling experience and streamline the design process. Especially features that result in a faster process or more clarity could be useful. This could include features like

- coping nodes and node groups,
- displaying multiple nets at once,
- loading multiple nets or rules at once,
- a more elaborate management interface for local rules.

The dynamic graph drawing algorithm of ReConNet, also still has some issues. Especially in larger nets with a higher number of nodes, after a net transformation many nodes of the net seem to cluster on the borders of the Petri net. Although [Lor17a] tried to enhance the graph drawing algorithm of ReConNet by introducing a node aging algorithm there is still room for improvement. Further development that results in a better distribution of Petri net nodes would increase the readability of ReConNets transformed Petri nets greatly.

Finally the model of substitution transitions can be extended to also include substitution places as well. While substitution transitions already seem to cover most hierarchy design solutions, exploring the use of places substitution could extend ReConNets design space further.

7. Conclusion

This thesis presents a hierarchical reconfigurable Petri net (HRPN) model based on substitution transitions that can be flattened into an equivalent non-hierarchical reconfigurable Petri net (RPN) model. For net verification and validation purposes the flat representation of the hierarchical reconfigurable Petri net can be used. A central part of this thesis is the formal definition of both the model and the flattening process in Chapter 3.

The HRPN model is then incorporated into a RPN modeling tool called ReConNet, in Chapter 4, thus extending ReConNets RPN capabilities to allow the modeling and simulation of HRPN as well.

The HRPN model and ReConNet tool are then evaluated using a typical use case in the form of a flexible manufacturing system (FMS). For this, in Chapter 5, a simple FMS is introduced and modeled using ReConNet and one additional similar tool. The second tool is called CPN tools and it is capable of hierarchical Petri net (HPN) modeling and simulation.

Form the evaluation it can be concluded that the use of HRPN for the modeling of dynamic systems can be quite beneficial. A dynamic system can create a multitude of possible system configurations during its operation and switch between these configurations on the fly. Which is an aspect that a model without reconfigurability struggles with. The reconfigurability aspect of the (H)RPN allows to unite all system configuration into one model, which enables the simulation and evaluation of the dynamic system as a whole.

The addition of hierarchy results in a clearer system model and significantly improves the reusability of reoccurring system parts. Which becomes more and more useful the larger the system becomes. The black box behavior of subnets and their local rules also eases the design process and modeling labor distribution.

The evaluation also reveals that the HRPN modeling capabilities added to ReConNet leave room for improvement. ReConNet especially lacks some features that similar HPN tools provide that would result in a more convenient and more expeditious modeling process. Future advancements, that are highlighted in Chapter 6, could close this gap.

Bibliography

[BCGS04] Stefano Basagni, Marco Conti, Silvia Giordano, and Ivan Stojmenovic. *Mobile ad hoc networking*. John Wiley & Sons, 2004.

[BCVH+03] Jonathan Billington, Søren Christensen, Kees Van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri net markup language: concepts, technology, and tools. In *International Conference on Application and Theory of Petri Nets*, pages 483–505. Springer, 2003.

[Chr13] George Chryssolouris. *Manufacturing systems: theory and practice*. Springer Science & Business Media, 2013.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation. (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.

[EHP+07] Hartmut Ehrig, Kathrin Hoffmann, Julia Padberg, Ulrike Prange, and Claudia Ermel. Independence of net transformations and token firing in reconfigurable place/transition systems. In *ICATPN*, volume 7, pages 104–123. Springer, 2007.

[EHP09] Hartmut Ehrig, Frank Hermann, and Ulrike Prange. Cospan DPO approach: An alternative for DPO graph transformations. *Bulletin of the EATCS*, pages 139–146, 2009.

[HHL+12] Monika Heiner, Mostafa Herajy, Fei Liu, Christian Rohr, and Martin Schwarick. Snoopy–a unifying Petri net tool. In *International Conference on Application and Theory of Petri Nets and Concurrency*, pages 398–407. Springer, 2012.

[HiP17] HiPS : Hierarchical Petri net Simulator. https://sourceforge.net/projects/hips-tools/, 2017.

[JK09] Kurt Jensen and Lars M Kristensen. *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009.

[JLL07] Gabriel Juhás, Fedor Lehocki, and Robert Lorenz. Semantics of Petri nets: A comparison. In *Simulation Conference, 2007 Winter*, pages 617–628. IEEE, 2007.

[JR12] Kurt Jensen and Grzegorz Rozenberg. *High-level Petri nets: theory and application*. Springer Science & Business Media, 2012.

[KJZJ08] Radek Koci, Vladimir Janousek, and Frantisek Zboril Jr. Object oriented Petri nets modelling techniques case study. In *Computer Modeling and Simulation, 2008. EMS'08. Second UKSIM European Symposium on*, pages 165–170. IEEE, 2008.

[KKR08] Hans-Jorg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. Graph transformation units–an overview. *Lecture Notes in Computer Science*, 5065:57–75, 2008.

[KV10] Peter Kostal and Karol Velisek. Flexible manufacturing system. *World Academy of Science, Engineering and Technology*, 53:825–829, 2010.

[LGB18] Learn and grow blog. http://www.learnandgrow.in/2015/05/fms-layout-types.html, 2018.

[Lor17a] Jan-Uriel Lorbeer. Dynamic graph drawing and conception of reconfigurable hierarchical Petri nets for ReConNet. DOI: 10.13140/RG.2.2.24513.66405, 2017.

[Lor17b] Jan-Uriel Lorbeer. Reconfigurable hierarchical Petri nets for ReConNet. DOI: 10.13140/RG.2.2.12769.61284, 2017.

[LP18] Jan-Uriel Lorbeer and Julia Padberg. Hierarchical, reconfigurable Petri nets. In *Workshops at Modellierung*, 2018.

[LWL$^+$06] Vincent Bardina Liu, Donald Louis Wires, Michael Joseph Lamping, Albert Michael Fischer, and Gary Lee Miller. Flexible manufacturing system, January 31 2006. US Patent 6,990,715.

[MK05] Toshiyuki Miyamoto and Sadatoshi Kumagai. A survey of object-oriented Petri nets and analysis methods. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 88(11):2964–2971, 2005.

[MM88] José Meseguer and Ugo Montanari. Petri nets are monoids: A new algebraic foundation for net theory. In *Logic in Computer Science, 1988. LICS'88., Proceedings of the Third Annual Symposium on*, pages 155–164. IEEE, 1988.

[MP12] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification.* Springer Science & Business Media, 2012.

[MRA10] V. Malhotra, T. Raj, and A. Arora. Excellent techniques of manufacturing systems: RMS and FMS. *International Journal of Engineering Science and Technology*, 2(3):137–142, 2010.

[Pad12] Julia Padberg. Abstract interleaving semantics for reconfigurable Petri nets. *Electronic Communications of the EASST*, 51, 2012.

[Pad15] Julia Padberg. Reconfigurable Petri nets with transition priorities and inhibitor arcs. In *International Conference on Graph Transformation*, pages 104–120. Springer, 2015.

[PEHP08] Ulrike Prange, Hartmut Ehrig, Kathrin Hoffmann, and Julia Padberg. Transformations in reconfigurable place/transition systems. *Lecture Notes in Computer Science*, 5065:96–113, 2008.

[PEOH12] Julia Padberg, Marvin Ede, Gerhard Oelker, and Kathrin Hoffmann. ReConNet: a tool for modeling and simulating with reconfigurable place/transition nets. *Electronic Communications of the EASST*, 54, 2012.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[PH15] Julia Padberg and Kathrin Hoffmann. A survey of control structures for reconfigurable Petri nets. *Journal of computer and communications*, 3(02):20, 2015.

[RE97] G. Rozenberg and H. Ehrig. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations. Vol. 1.* 1997.

[Ren03] Arend Rensink. The groove simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

[RPL$^+$08] Alexander Rein, Ulrike Prange, Leen Lambers, Kathrin Hoffmann, and Julia Padberg. Negative application conditions for reconfigurable place/transition systems. *Electronic Communications of the EASST*, 10, 2008.

[RWL$^+$03] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and

Kurt Jensen. CPN tools for editing, simulating, and analysing coloured Petri nets. In *International Conference on Application and Theory of Petri Nets*, pages 450–462. Springer, 2003.

[SCDB14] Pengfei Sun, Simon Collart-Dutilleul, and Philippe Bon. A formal modeling methodology of the french railway interlocking system via HCPN. *WIT Transactions on The Built Environment*, 135:849–858, 2014.

[Sch14] Alexander Schulz. Model checking of reconfigurable Petri nets. *arXiv preprint arXiv:1409.8404*, 2014.

[Sti05] Marc Stiegler. Petname systems. *HP Laboratories, Mobile and Media Systems Laboratory, Palo Alto, Tech. Rep. HPL-2005-148*, 2005.

[TA15] Weng Jie Thong and M.A. Ameedeen. A survey of Petri net tools. In *Advanced Computer and Communication Engineering Technology*, pages 537–551. Springer, 2015.

[Tae99] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 481–488. Springer, 1999.

[Tet90] Ulrich A. W. Tetzlaff. Flexible manufacturing systems. In *Optimal Design of Flexible Manufacturing Systems*, pages 5–11. Springer, 1990.

[TPCS12] Bogdan Târnaucă, Dan Puiu, Vasile Comnac, and Constantin Suciu. Modelling a flexible manufacturing system using reconfigurable finite capacity Petri nets. In *Optimization of Electrical and Electronic Equipment (OPTIM), 2012 13th International Conference on*, pages 1079–1084. IEEE, 2012.

[Zim08] Armin Zimmermann. Colored Petri nets. *Stochastic Discrete Event Systems: Modeling, Evaluation, Applications*, pages 99–124, 2008.

[ZZ09] Yong Zhang and Jing Zhu. Product line system modeling of the cold-rolled mill based on the hierarchy colored Petri nets. In *Automation and Logistics, 2009. ICAL'09. IEEE International Conference on*, pages 1553–1557. IEEE, 2009.

A. Petri Nets and Rules of the FMS Evaluation Example

Main Net

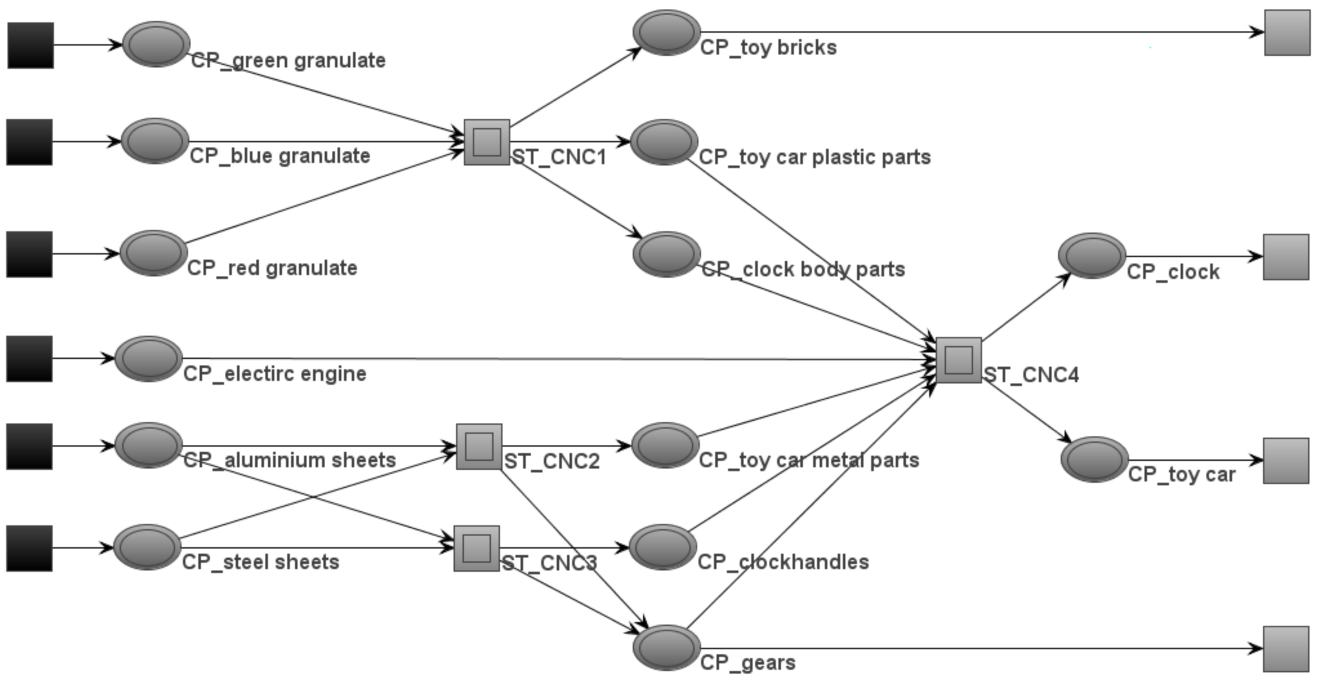


Figure A.1.: FMS evaluation example: main net

CNC1 - Injection molder

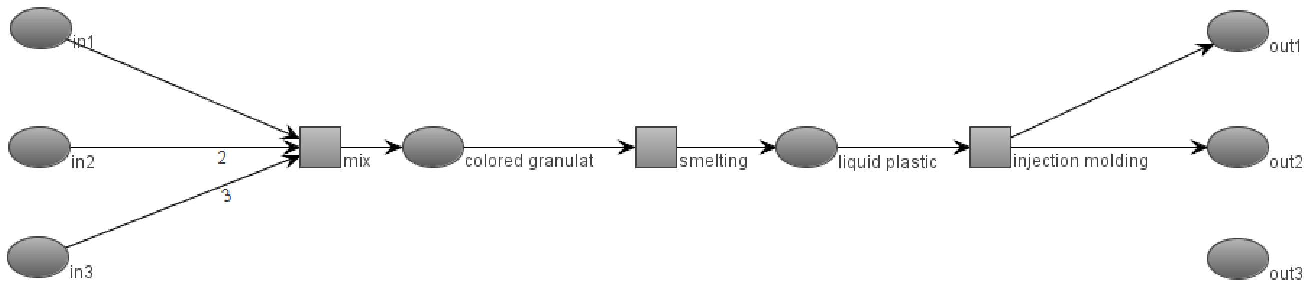


Figure A.2.: FMS evaluation example: injection molder template net.

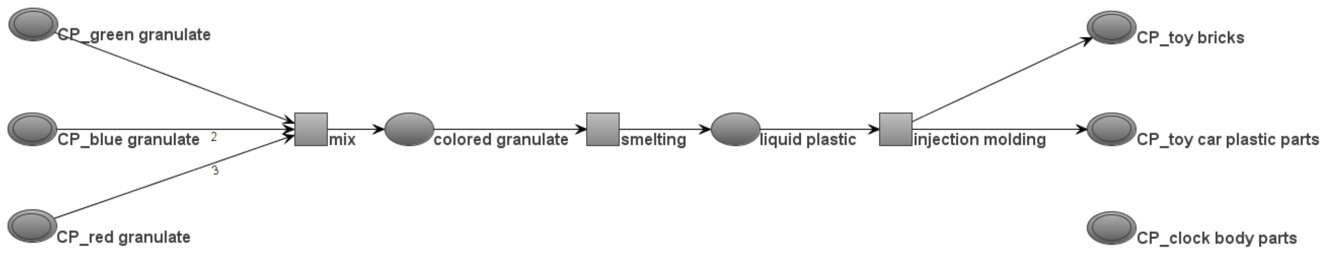


Figure A.3.: FMS evaluation example: subnet CNC1

The Local Rules

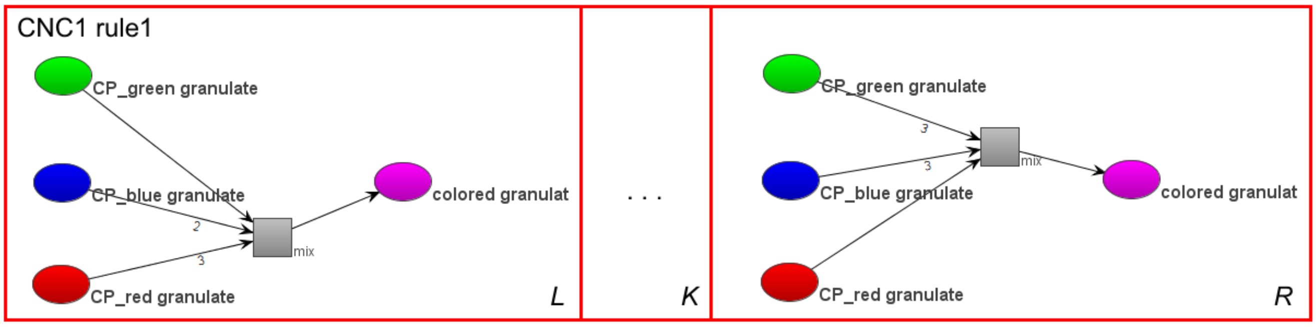


Figure A.4.: FMS evaluation example: rule CNC1R1

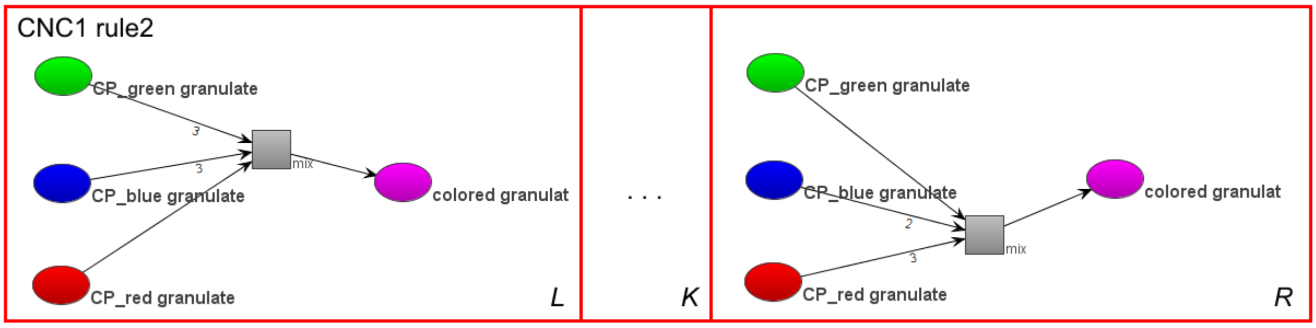


Figure A.5.: FMS evaluation example: rule CNC1R2

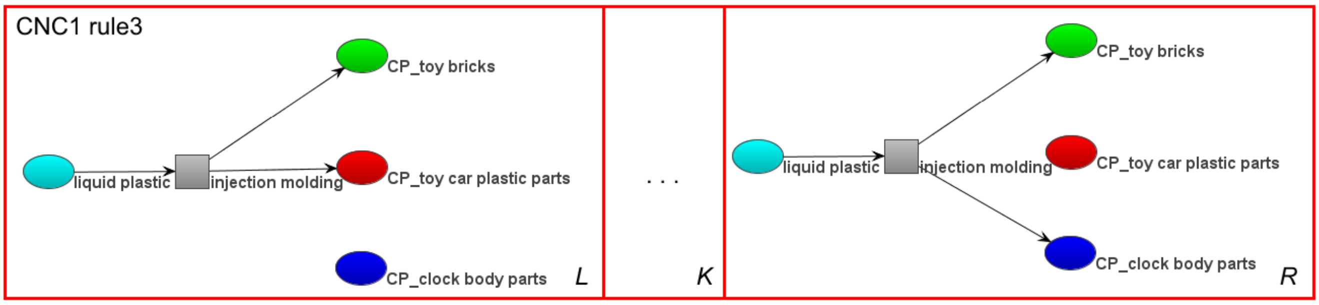


Figure A.6.: FMS evaluation example: rule CNC1R3

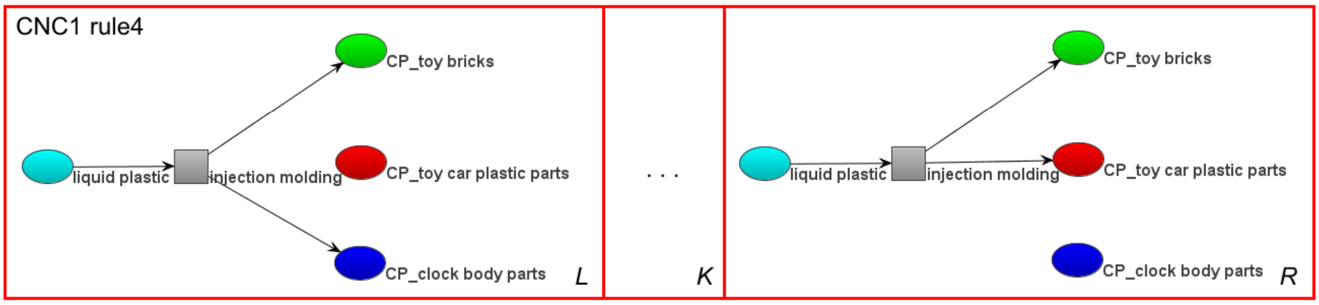


Figure A.7.: FMS evaluation example: rule CNC1R4

CNC2 & 3 - Laser Cutter1 and 2

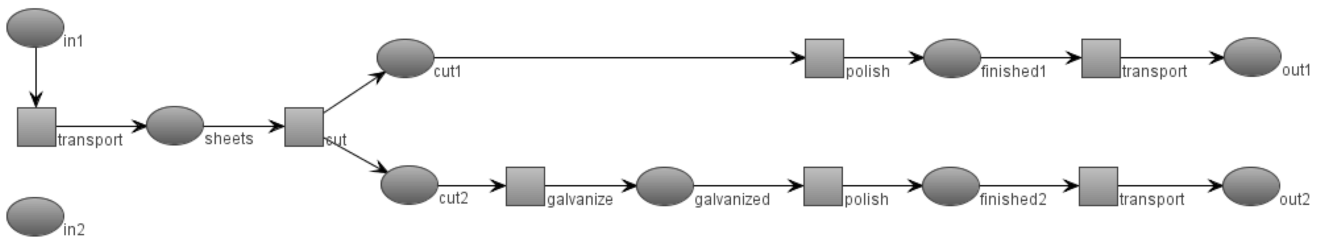


Figure A.8.: FMS evaluation example: laser cutter template net.

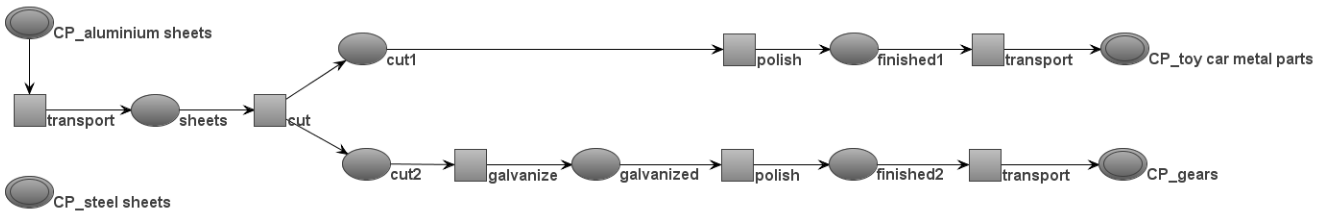


Figure A.9.: FMS evaluation example: subnet CNC2

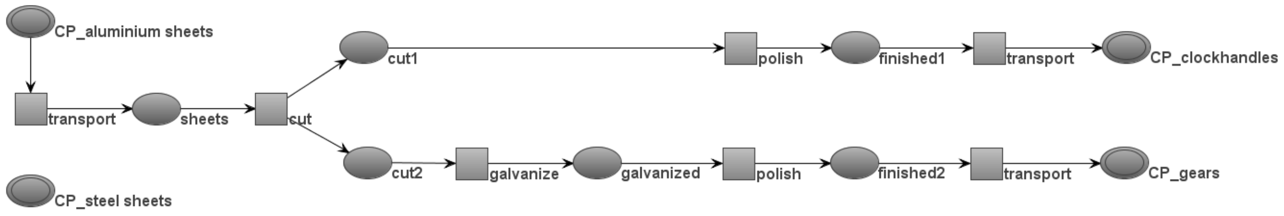


Figure A.10.: FMS evaluation example: subnet CNC3

The Local Rules

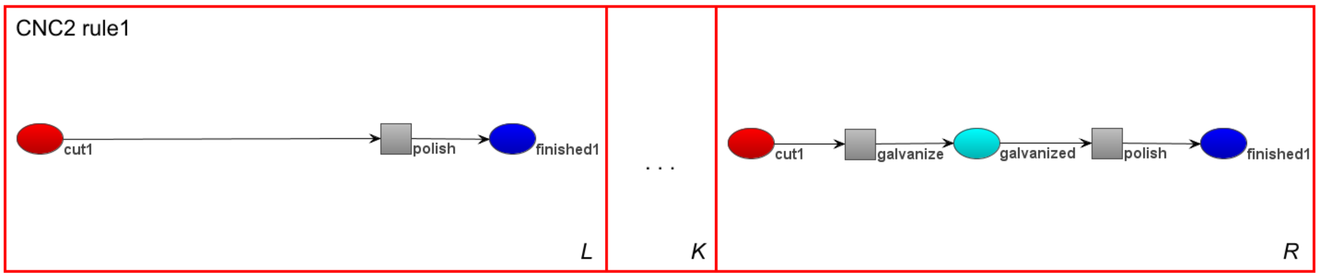


Figure A.11.: FMS evaluation example: rule CNC2R1

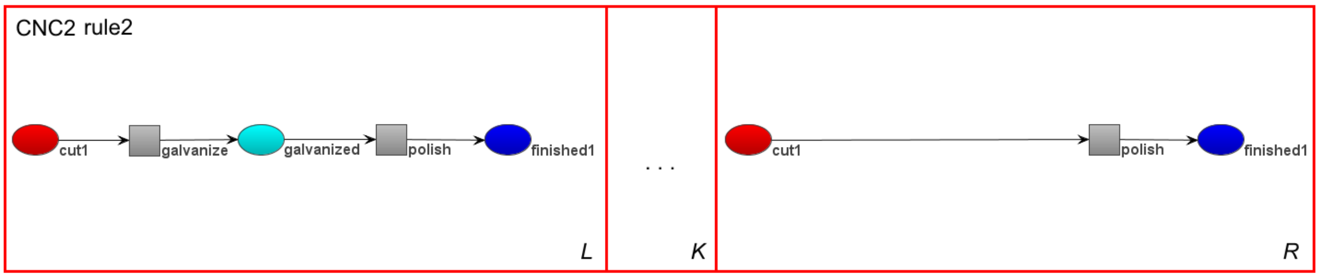


Figure A.12.: FMS evaluation example: rule CNC2R2

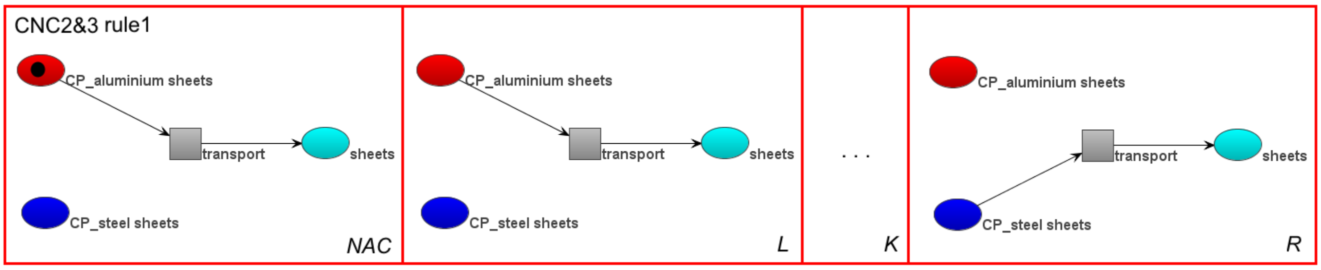


Figure A.13.: FMS evaluation example: rule CNC23R1

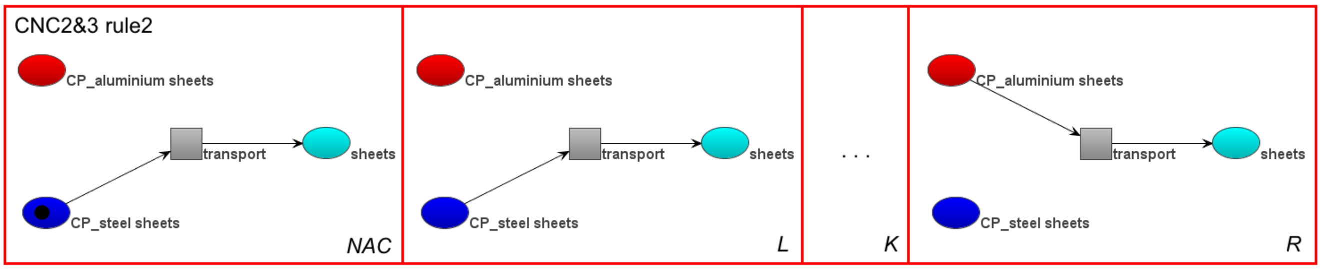


Figure A.14.: FMS evaluation example: rule CNC23R2

CNC4 - Assembly Machine

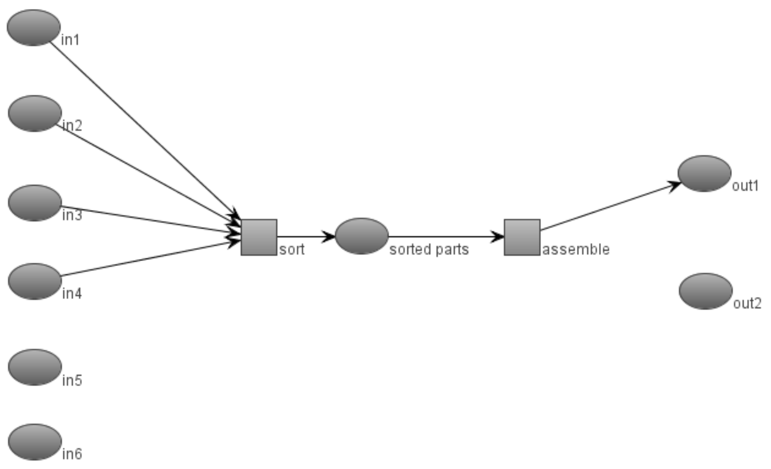


Figure A.15.: FMS evaluation example: assembly machine template net.

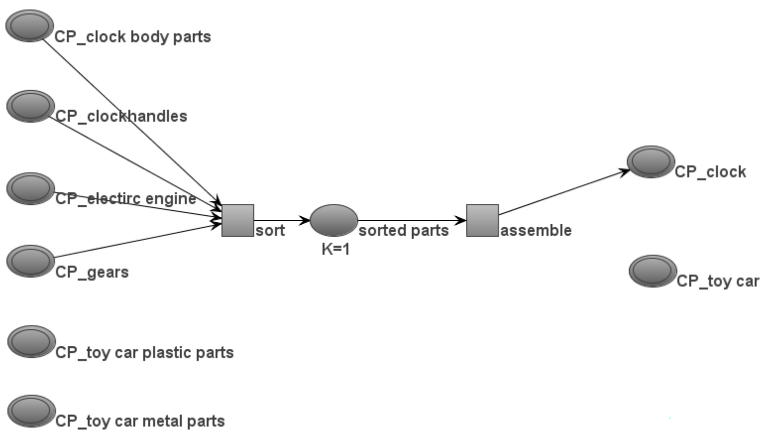


Figure A.16.: FMS evaluation example: subnet CNC4

The Local Rules

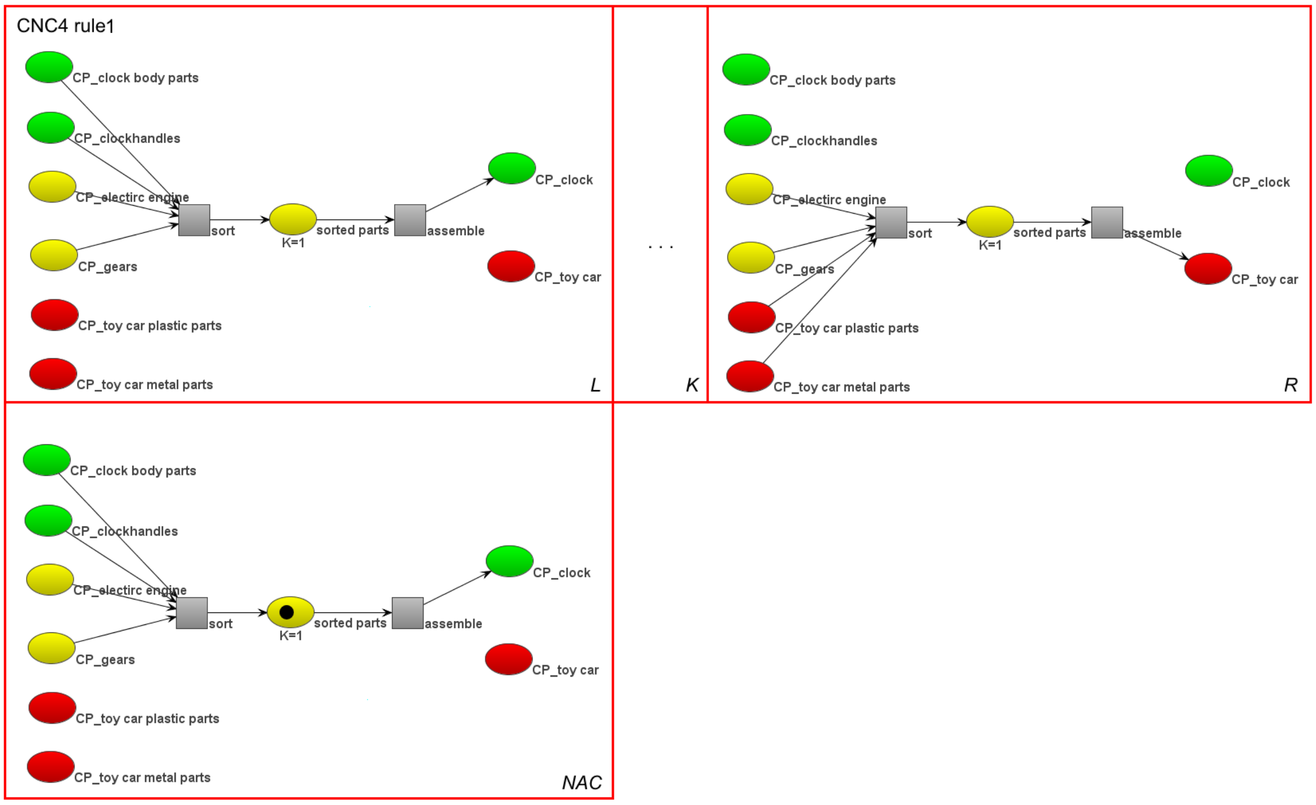


Figure A.17.: FMS evaluation example: rule CNC4R1

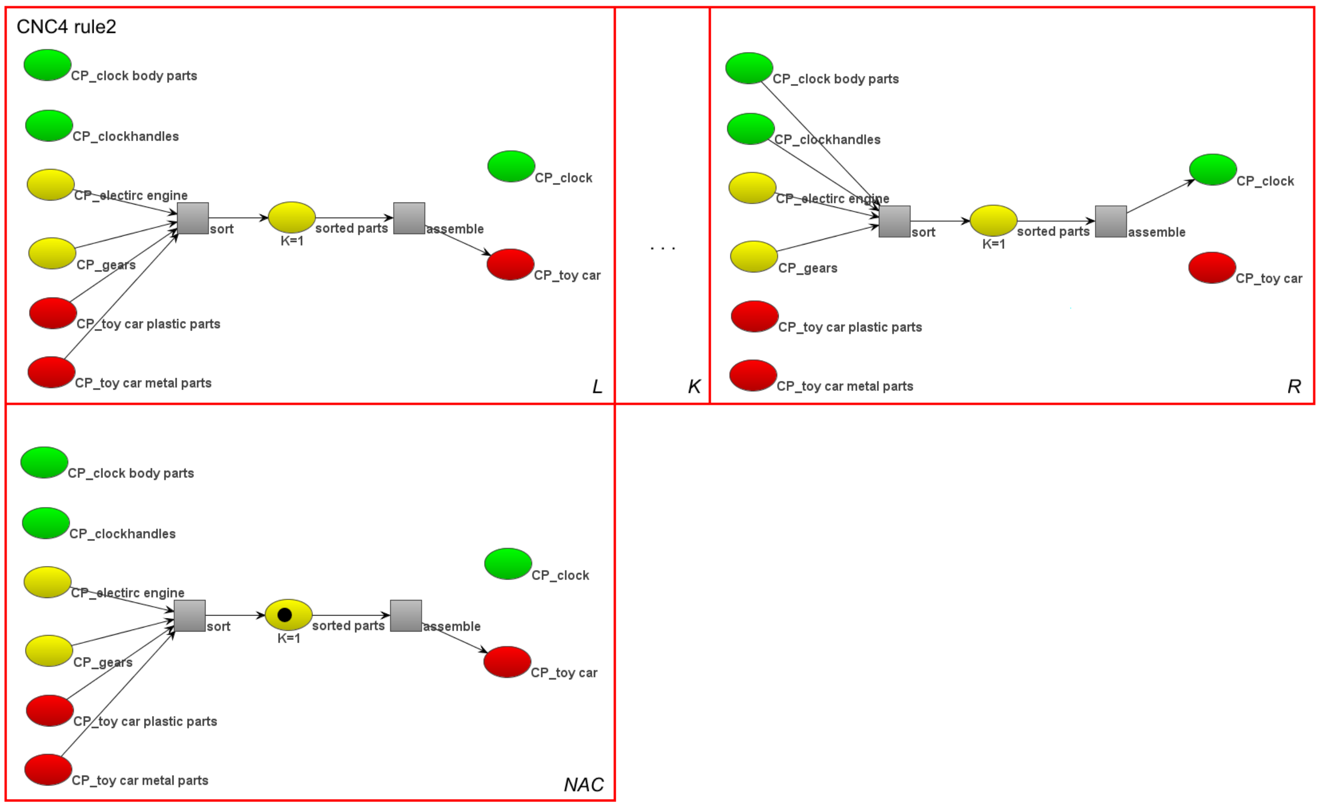


Figure A.18.: FMS evaluation example: rule CNC4R2

Flat Net

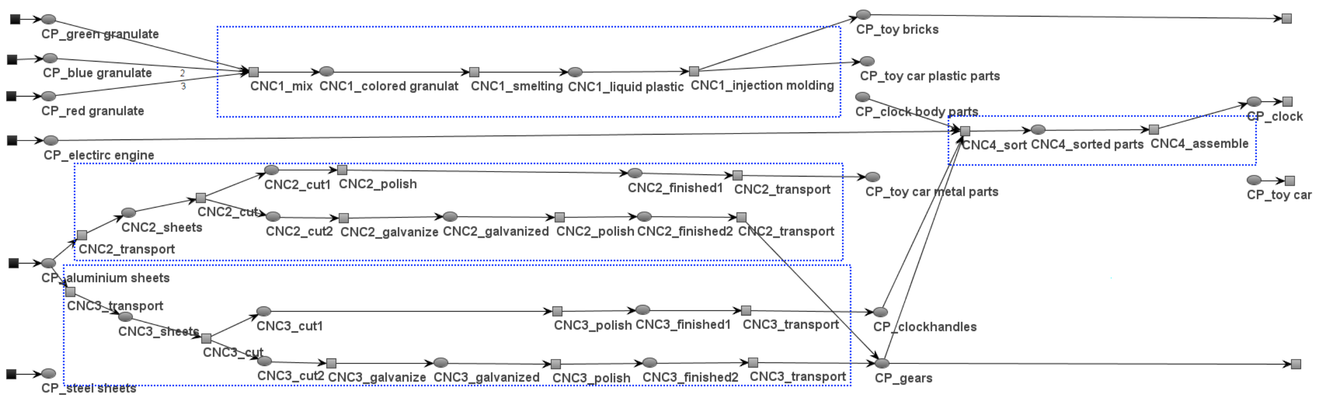


Figure A.19.: FMS evaluation example: flat net. Former subnets are marked in blue.

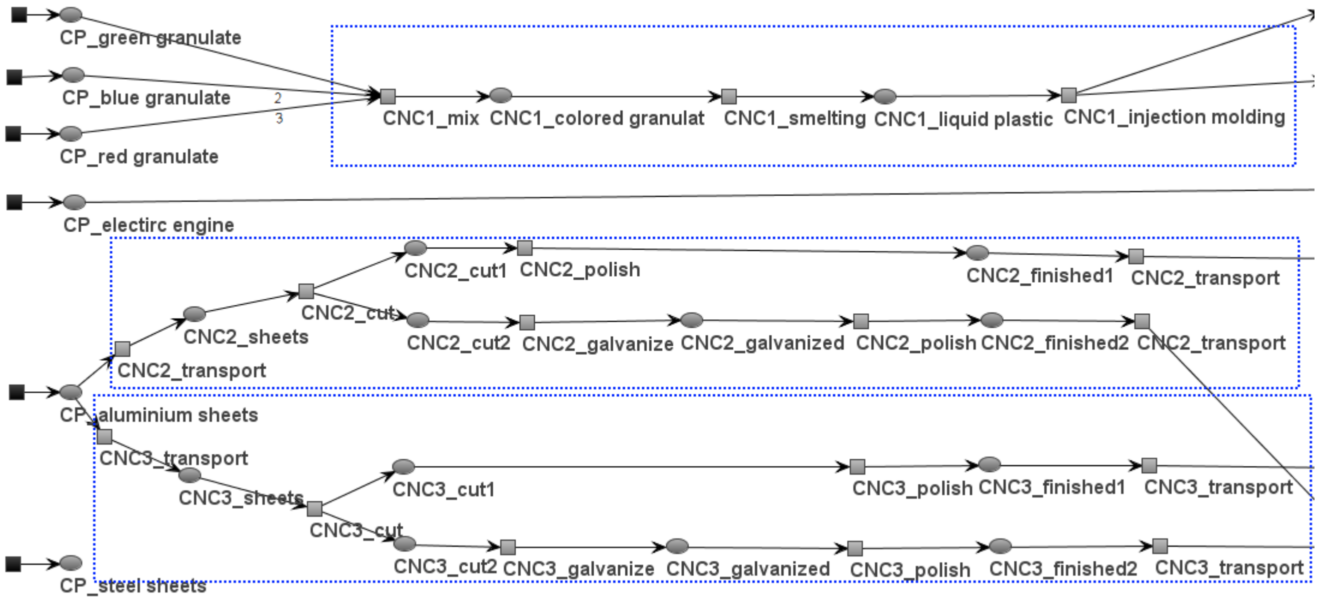


Figure A.20.: FMS flat net left side.

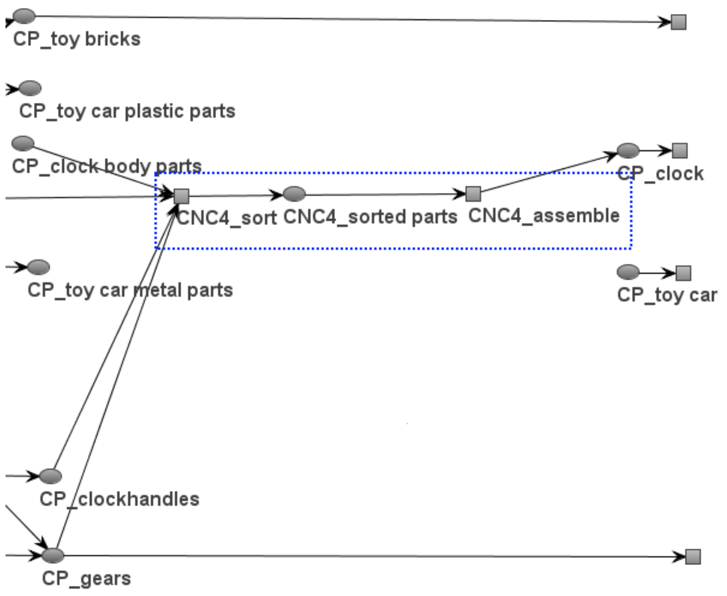


Figure A.21.: FMS flat net right side.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28.05.2018 Jan-Uriel Lorbeer