

Automatic Modularization of Place/Transition Nets

Julian Gaede¹, Judith-Henrike Overath¹ and Sophie Wallner¹

¹Universität Rostock, 18051 Rostock, Germany (<firstname.lastname>@uni-rostock.de)

Abstract

Modular Petri nets provide a way to alleviate the state-space explosion problem. However, Petri nets are usually not described as a modular structure. In order to benefit from the advantages of modular state space analysis, the Petri net has to be decomposed into modules in advance. In this paper, we discuss various approaches to automatic modularization. One approach focuses on finding replicated modules in a Petri net, since the state space for those replications needs to be computed only once during analysis. An implementation based on the graph isomorphism problem shows the validity of this idea. The effectiveness of state space reduction by modularization is significantly influenced by the size of the individual modules and the size of the interfaces between them. If possible, the modules should exhibit internal behavior, i.e. have a certain minimum size and produce nontrivial strongly connected components in their reachability graphs. The interfaces should be as small as possible to avoid an overhead of synchronization behavior at the interfaces. To this end, we firstly present an approach that produces modules with meaningful internal behavior based on T-invariants. Secondly, we explain two approaches that use hypergraph cuts to minimize the size of interfaces.

Keywords

Petri Nets, Modular State Space, Replication, Modularization,

1. Introduction

A modular Petri net consists of subnets that act independently of each other and are synchronized via interfaces. Modularization is the concept of transforming a Petri net system into an equivalent modular Petri net with independent components but common interfaces. Sometimes Petri net system models describe systems that consist of such components anyway. However, this encapsulation is not represented by the conventional Petri net system formalism. Modularization offers the possibility to make this information exploitable for analysis. Another application is the decomposition of models that would not be expected to have a modular structure. Especially for Petri net systems with identical structures, modularization provides a suitable solution as it is more robust with respect to small deviations in contrast to symmetry. Regarding the well-known example of the dining philosophers, the symmetry is broken as soon as one of the philosophers grabs the forks in the opposite order. Anyway, such a version of the dining philosophers can be modularized according to our methods.

The idea of decomposing a Petri net system for analysis is not new. One way of forming components is clustering [1]. However, the clusters do not have to be disjoint and therefore do not act independently of each other. There is also the approach of forming a compositional state space [2]. Here, the state spaces of the individual Petri net system components are calculated individually and then a global state space of the entire Petri net system is derived. Since the partial state spaces must be completely available for further calculations, the components must be bounded. Modularization does not require this restriction; components can be unbounded in themselves and only become restricted through their embedding into a modular Petri net. In this paper, Section 2 first introduces all the necessary concepts in the domain of modular Petri nets, as the notation differs from the notation of conventional Petri net systems. Then, we continue exploring the issue of automatic modularization of Petri net systems with a focus on increasing efficiency in the analysis. A major advantage in the analysis of a modular state space arises from the fact that the original Petri net system might contain replicated components, for which it is sufficient to compute the state space once [3]. Therefore, Section 3 concentrates on finding replications in the Petri net. As a Petri net system is basically a graph, the idea of finding replications can be reduced to finding isomorphic subgraphs in a graph. An implementation shows the validity of

PNSE'24, International Workshop on Petri Nets and Software Engineering, 2024



© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

this approach. Section 4 presents further ideas on how to modularize a Petri net system; especially when no more replications are expected. This is the case, for example, when all replications have already been found in a Petri net system, but a sufficiently large part of the net is still unmodularized. Likewise, the ideas can also be applied to modularize replications that have already been found, for example, if they are still quite large. Here, however, modularization should not simply be done blindly, but based on the structure of the Petri net system. One approach makes use of transition invariants, which are considered an important indicator of nontrivial behavior. Thus, a modularization based on t-invariants is expected to generate modules with meaningful behavior. What impedes the analysis in the modularized Petri net is the synchronization behavior of the modules among each other. Therefore, it is a legitimate reason to want to keep this synchronization behavior rare. This can be achieved by modularizing Petri net system in a way that the interfaces between modules are as small as possible. For this, (hyper)graph theoretical concepts are consulted.

2. Preliminaries

Definition 1 (Petri net system). A *Petri net system* is a tuple $N = [P, T, F, W, m_0]$, where

- P is a finite set of *places*,
- T is a finite set of *transitions* with $P \cap T = \emptyset$,
- $F \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs*,
- $W : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ is the *weight function* with $W(x, y) = 0$ iff $(x, y) \notin F$ and
- m_0 is the *initial marking*, where a marking in general is a mapping $m : P \rightarrow \mathbb{N}$.

We call $P \cup T$ the *nodes* of a Petri net system. For a node $x \in P \cup T$, $\bullet x = \{y \mid (y, x) \in F\}$ the *preset* of x and $x \bullet = \{y \mid (x, y) \in F\}$ the *postset* of x . We call $\bullet x \cup x \bullet$ the *environment* for a node $x \in P \cup T$. The objective is to generate a modular structure for a given Petri net system, such that the modular Petri net generated out of the structure is isomorphic to the given system. Therefore, we introduce concepts for describing a modular structure. We follow the notation established in [3].

Definition 2 (Module). A *module* is a place/transition Petri net $N_i = [P_i, T_i, F_i, W_i]$ for $i \in \mathbb{N}$ where the set T_i is assumed to be partitioned into subsets $T_{i|internal}$ of internal transitions and $T_{i|interface}$ of interface transitions.

A module describes some part of the full model's structure. To exhibit any behavior, a module needs to be *instantiated* by adding an initial marking.

Definition 3 (Instance). An *instance* is a Petri net system $[N_i, m_{0i}]$ of module N_i for $i \in \mathbb{N}$, where $m_{0i} : P_i \rightarrow \mathbb{N}$ is an initial marking of N_i .

While each instance has some self-contained (or *internal* behavior), they may synchronize with other instances via their interface transitions. This synchronization is accomplished by combining interface transitions of different instances to a fusion transition to force them to fire simultaneously if all individual preconditions are met. The fusion is guided by fusion vectors.

Definition 4 (Fusion Vector, Support). Let $\mathcal{I} = \{[N_1, m_{01}], \dots, [N_\ell, m_{0\ell}]\}$ be a set of instances. A *fusion vector* $f \in (T_{1|interface} \cup \{\perp\}) \times \dots \times (T_{\ell|interface} \cup \{\perp\})$ is a vector of interface transitions of the instances or \perp . An instance $[N_j, m_{0j}]$ participates in f with its interface transition $t \in T_{j|interface}$ if $f[j] = t$. If $f[j] = \perp$, the instance does not participate in the fusion. The *support* $supp(f) = \{t \mid f[j] = t\}$ of a fusion vector is the (nonempty) set of contained interface transitions.

An instance can participate at most once in a given fusion, but the same interface transition can appear in multiple fusion vectors.

We are now ready to collect all information needed for the composition of modules.

Definition 5 (Modular Structure). A *modular structure* is a tuple $\mathcal{M} = [\mathcal{I}, \mathcal{F}]$, where $\mathcal{I} = \{[N_1, m_{01}], \dots, [N_\ell, m_{0\ell}]\}$ is a set of instances of pairwise disjoint modules and $\mathcal{F} \subseteq (T_{1|interface} \cup \{\perp\}) \times \dots \times (T_{\ell|interface} \cup \{\perp\})$ is a set of fusion vectors.

From this modular structure we can derive a modular Petri net system, where places, internal transitions, and initial markings are taken from the individual instances. Every fusion vector defines another transition that inherits all the arcs of the contained interface transitions and establishes connections across instances. Consequently, interface transitions that do not appear in any fusion set do not appear in the composition. This can be trivially worked around by adding a fusion vector with only this transition and \perp everywhere else.

Definition 6 (Modular Petri Net, Fusion Transition). Let $\mathcal{M} = [\mathcal{I}, \mathcal{F}]$ be a modular structure. From \mathcal{M} , we can derive a Petri net system $N = [P, T, F, W, m_0]$, where

- $P = \bigcup_{j \in \{1, \dots, \ell\}} P_j$,
- $T = \bigcup_{j \in \{1, \dots, \ell\}} T_{j|internal} \cup \{t_f \mid f \in \mathcal{F}\}$, where t_f is the *fusion transition* for $f \in \mathcal{F}$,
- $F = \bigcup_{j \in \{1, \dots, \ell\}} (F_j \cap (P_j \times T_{j|internal} \cup T_{j|internal} \times P_j)) \cup \{(p, t_f) \mid f \in \mathcal{F}, (p, f[j]) \in F_j\} \cup \{(t_f, p) \mid f \in \mathcal{F}, (f[j], p) \in F_j\}$
- $W(t, p) = \begin{cases} W_j(t, p) & \text{for } (t, p) \in F_j \text{ and } t \in T_{j|internal} \\ W_j(t^*, p) & \text{for } (t^*, p) \in F_j \text{ and } t^* \in \text{supp}(f), \\ & f \in \mathcal{F}, t = t_f, j \in \{1, \dots, \ell\} \end{cases}$
- $W(p, t) = \begin{cases} W_j(p, t) & \text{for } (p, t) \in F_j \text{ and } t \in T_{j|internal} \\ W_j(p, t^*) & \text{for } (p, t^*) \in F_j \text{ and } t^* \in \text{supp}(f), \\ & f \in \mathcal{F}, t = t_f, j \in \{1, \dots, \ell\} \end{cases}$
- $m_0 = \bigcup_{j \in \{1, \dots, \ell\}} m_{0j}$

We call N the *modular Petri net* for \mathcal{M} .

Note that the initial marking of the modular Petri net is well-defined since the domains of all m_{0j} are pairwise disjoint.

To generate a modular structure from a non-modular Petri net, we only need to find a mapping that unambiguously assigns an instance to each place, thus we get a place set for every instance. The other components of the modular structure, from which we can deduce a modular Petri net, can be derived from this set of places as shown in the following lemma.

Lemma 1 (Deducing a modular structure from a partitioning of places). Let $N = [P, T, W, F, m_0]$ be a Petri net. Further, let $\{P_1, \dots, P_\ell\}$ be a partitioning of P . For P_j with $j \in \{1, \dots, \ell\}$, we generate instance $[N_j, m_{0j}]$ with

- $T_j = \bigcup_{p \in P_j} \{(\bullet p \cup p \bullet) \times j\}$
- $((t, j), p) \in F_j$, iff $(t, p) \in F$; $(p, (t, j)) \in F_j$, iff $(p, t) \in F$
- $W_j((t, j), p) = W(t, p)$; $W_j(p, (t, j)) = W(p, t)$ for a $p \in P_j$
- $m_{0j} = m_0 \cap (P_j \times \mathbb{N})$

We append the index j to each transition to distinguish their occurrences in different instances. The fusion vectors can be constructed by identifying transitions which occur in multiple instances. For $t \in T$, we identify the modules where t occurs as $indices(t) = \{j \mid (t, j) \in T_j\}$. Then we generate a fusion vector f for all t where $|indices(t)| > 1$, so that $f[j] = (t, j)$ if $j \in indices(t)$ and \perp otherwise. We can distinguish internal and interfaces transitions for the instances as well. Therefore, we declare $(t, j) \in T_{j|interface}$, iff $\exists f \in \mathcal{F} : (t, j) \in f$ and $T_{j|internal} = T_j \setminus T_{j|interface}$. This results in a set of instances $\mathcal{I} = \{[N_1, m_{01}], \dots, [N_\ell, m_{0\ell}]\}$.

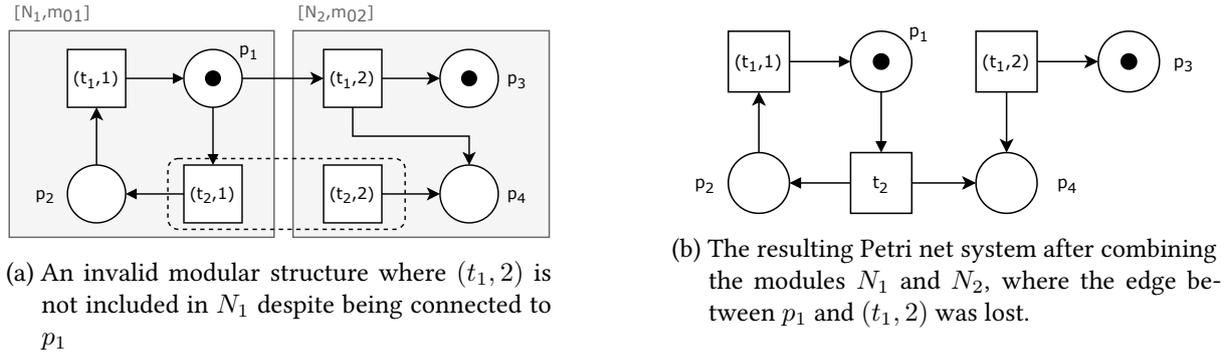


Figure 1: An invalid modularization resulting in the loss of structure after combining the modules.

Example 1. Figure 1 illustrates why it is required to include all connected transitions for a given place into the same module. The place p_1 is included in module N_1 and connected to $(t_{1, 2})$, which is in another module. Since the composition rules for a modular Petri net [3] only concern themselves with fusion transitions, in this case $(t_{2, 1})$ and $(t_{2, 2})$, which are fused into t_2 in the resulting net. The connection between p_1 and $(t_{1, 2})$ is not reconstructed, changing the behavior of the modular Petri net compared to the composition of the modules.

3. Modularization based on Replications

It is desirable to allow a single module to be *replicated*, or instantiated multiple times with possibly different initial markings, since this makes sense from a modelling perspective, but also unlocks additional memory-savings during model checking. To distinguish between those replications, we extend instances by a unique identifier, and call them *replicated instances* or short: *r-instances*.

Definition 7 (R-Instance). An *r-instance* is a Petri net system $[N_i, j, m_{0j}]$ of module N_i , where $j \in \mathbb{N}$ is an instance identifier and $m_{0j} : P_i \rightarrow \mathbb{N}$ is an initial marking of N_i .

The definitions for the modular structure, fusion vectors and modular Petri net can be easily amended to accommodate this extension. A fusion vector can contain transitions from multiple *r-instances* of the same module, since the module would be copied for each instance when constructing the modular Petri net.

To identify replicated modules, we represent the net structure of the given Petri net system as a net graph. The marking of places is irrelevant for this, as replicated modules with different markings can still lead to a reduction in the local reachability graph compared to the combined size of individual modules[3].

Definition 8 (Net graph). For a given Petri net $N = [P, T, F, W, m_0]$, the *net graph* $G = [V, E, c]$ is a directed, labeled graph, where

- $V = P \cup T$
- $E = F$
- $c(v) = p$ if $v \in P$
- $c(v) = t$ if $v \in T$
- $c((v, v')) = W(v, v')$

In the way we define a modular structure, the state space of instances of the same module (regardless the initial marking) can be analyzed based on one common state structure. In general, identical structures lead to identical behavior, which we do not need to analyze repeatedly. Therefore, it makes sense to identify identical substructures in the net graph, which we then can declare as instances of one module.

Given a net graph, the ideal scenario would be to maximize the number of identical structures, i.e. isomorphic induced subgraphs. Those subgraphs should be larger than a minimum size to avoid the trivial solution where each vertex is a subgraph. This problem is germane to the INDUCED SUBGRAPH ISOMORPHISM problem or the more general MAXIMUM COMMON INDUCED SUBGRAPH problem, where for two given graphs one must determine whether one graph is an induced subgraph of the other resp. the two graphs have a common induced subgraph with a minimum size. Those problems are known to be NP-complete, as described in [4]. The problem might as well be associated with the LARGEST SUBGRAPH WITH A PROPERTY problem [4], where the task is to find the maximum subgraph with a given property. Considering the repeated occurrence of a subgraph as a property does not serve as a standard graph property why this problem can be neglected. From the field of data mining, the challenge to find identical substructures in graphs is known as the *frequent subgraph mining* [5]. Expressing our problem as a frequent induced subgraph mining problem turned out to be unsuitable. Frequent subgraph mining in general does not consider induced subgraphs, which is mandatory for us [6]. There are also approaches for induced subgraphs but on the base of multiple graphs [7].

Another related field concerns itself with the identification of overrepresented sub-structures in a network, so-called *motifs*. The algorithms from this field can be broadly categorized into *motif-centered* and *network-centered* approaches. A motif-centered procedure requires a motif as part of its input, which makes it unsuitable for our application. Conversely, a network-centered method starts by enumerating all subgraphs for a given size and counts their occurrence in the graph [8]. However, the performance of these approaches tends to scale poorly with the size of the motifs. Since the state space reduction achieved by the modularization of a Petri net system is heavily impacted by the amount of internal behavior of its modules [3], it is desirable to identify larger substructures. Therefore, a network-centered approach isn't appropriate either to find replicated modules with sufficient internal behavior.

Besides maximizing the number of identical structures we impose additional constraints to a solution. When considering subgraphs of Petri nets, not every induced subgraph is a valid replication. For every place we require all connected transitions to be included in the instance of the place. Otherwise, the edges of an omitted transition to a place cannot be reconstructed in the construction of the modular Petri net as described in Definition 6.

With the graph theoretical background of the suspected NP-completeness of our problem we developed an incremental heuristic approach to identify identical substructures in a net graph. The main idea is to unfold the substructures circularly from selected starting points while preserving isomorphism between them. Therefore, we present a necessary criterion and multiple heuristic concepts that lead to a more or less precise starting point selection. The different concepts provide various degrees of freedom.

3.1. Finding Seed Candidates

First, we need to identify potential candidates that can serve as starting points, or *seeds*, for detecting isomorphic substructures. Given the net graph G of a Petri net system N , we only consider place vertices, i.e. vertices with $c(v) = p$ as candidates. As described above, a place requires all adjacent transitions to be in its instance, while a transition does not impose such constraints. Considering the net graph of a given Petri net system we introduce the signature of a vertex, determining the number of neighbor vertices for a given edge type.

Definition 9 (Signature). For a vertex $v \in V$ we define the *signature* as a mapping $s : V \rightarrow (\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N})$, such that $s(v)(i, o) = |\{v' \mid c(v, v') = o, c(v', v) = i\}|$.

The signature gives an initial impression of the direct neighborhood of a vertex v , i.e. how many vertices are connected to v via a certain combination of incoming and outgoing edges.

Example 2. In the Petri net system presented in Fig. 1b, the places p_1 and p_2 would share the signature $s(p_i) = \{(1, 0) \mapsto 1, (0, 1) \mapsto 1\}$, while $s(p_3) = \{(1, 0) \mapsto 1\}$ and $s(p_4) = \{(1, 0) \mapsto 2\}$.

Only place vertices with identical signature can serve as starting points, since all neighboring transitions have to be included, which would immediately lead to non-isomorphic modules.

To determine the actual seeds using the information provided by the signature, it makes sense to consider the number of candidates for each signature. It appears that candidates that share an infrequent signature are more promising seed vertices, since vertices with a rare connectivity structure are more likely to have same function in the model. However, the fewer candidates share a signature, the harder it becomes to ensure a spacing between seeds which allows the modules to grow to a sufficiently large size before colliding with another.

While there exist methods to find a set of vertices in a graph with maximum distance between each other, it is in many cases sufficient to select vertices with a *large enough* distance between them. To define such a distance, one could look towards a specified maximum module size or attempt to exploit structural properties of the Petri net system. Once a minimum distance d_{min} has been found, suitable seeds can be identified by starting a simultaneous depth-limited BFS from all candidates with a shared signature. If the BFS from a candidate finishes without encountering another candidate's nodes within $d_{min}/2$ steps, this candidate can serve as a seed. Otherwise, a seed can be selected from each set of conflicting candidates. In the case that there are too few suitable candidates remaining, the requirements have to be relaxed, either by reducing d_{min} or by restarting the selection process with candidates with another shared signature.

Besides those systematic approaches, a randomized selection can be performed instead. While the candidates still have to share their signature, it can be beneficial to omit checking the distance between candidates and instead relying on the expansion algorithm to deal with the colliding modules.

Developing improved heuristics for seed selection is an area of further research, as well as quantifying their impact on the quality of the resulting modularization.

3.2. Expanding outwards

Regardless the method we determine seeds, we receive a set of n starting points suitable for determining isomorphic starting points in the Petri net system. To find sufficiently large replicated modules, we greedily expand the modules outwards from those starting points under a set of guidelines that will be presented in the following. During the process, each module is represented by a list R_i , which contains sets of vertices. Those sets will be called *shells* and contain the nodes which were newly added during each step.

The outline of this algorithm can be found in Algorithm 1, and will be subsequently defined by explaining the sub-procedures.

3.2.1. Finding expansion candidates

In each expansion step, we first calculate the maximum possible expansion step of each module and then successively reduce this step until either all conditions for replicated modules are met or a necessary reduction fails. Since the modules grow from center outwards, it is sufficient to consider only neighbors of vertices from the outermost shell of each module, which weren't already introduced in previous shells. Additionally, we maintain a set of vertices which are forbidden from being included or explored due to having caused conflicts in previous steps. Due to the bipartite nature of Petri nets, as well as the fact that we required each seed vertex v_i^S to be $\in P$, the shells created by this approach alternate between containing only place or transition vertices, but never a mix of both.

3.2.2. Resolving conflicts between steps

The first reduction of the expansion step is accomplished by detecting conflicts between the steps. In every odd step, the candidate shells only contain transitions, which are allowed to overlap between modules. While the transitions remain in the expansion step, we mark them as forbidden to explore further to prevent complications downstream, such as the conflicting modules attempting to explore along the same paths in future steps.

Algorithm 1 Algorithm to find replications

```

procedure FINDREPLICATIONS( $N, \{v_1^S, \dots, v_n^S\}, \text{maxsize}$ )
  global stop_nodes =  $\emptyset$  ▷ Set of nodes not to be explored further
   $R_1 \leftarrow \langle \{v_1^S\} \rangle, \dots, R_n \leftarrow \langle \{v_n^S\} \rangle$  ▷ Each replication consists of a sequence of steps
  repeat
     $\text{step}_1, \dots, \text{step}_n \leftarrow \text{CALCULATENEXTSTEPS}(N, \langle R_1, \dots, R_n \rangle)$ 
     $\text{step}_1, \dots, \text{step}_n \leftarrow \text{RESOLVECONFLICTS}(\langle R_1, \dots, R_n \rangle, \langle \text{step}_1, \dots, \text{step}_n \rangle)$ 
    if not all step sizes are equal then
       $\text{step}_1, \dots, \text{step}_n \leftarrow \text{EQUALIZESTEPS}(\langle \text{step}_1, \dots, \text{step}_n \rangle)$ 
    end if
     $\langle \sigma_1, \dots, \sigma_{n-1} \rangle \leftarrow \text{CALCULATEISOMORPHISM}(\langle R_1, \dots, R_n \rangle, \langle \text{step}_1, \dots, \text{step}_n \rangle)$ 
    if any of the previous operations fail then
      break
    end if
    for all  $i \in \{1, \dots, n\}$  do
       $R_i += \text{steps}_i$ 
    end for
  until modules exceed maxsize
  if  $|R_1| \bmod 2 = 0$  then ▷ Even radius = Outer shell contains only places
    remove last shell from  $R_1, \dots, R_n$ 
  end if
  return  $\langle R_1, \dots, R_n \rangle, \langle \sigma_1, \dots, \sigma_{n-1} \rangle$ 
end procedure

```

Algorithm 2 Subroutine to find calculate the next expansion step

```

procedure CALCULATENEXTSTEPS( $N, \langle R_1, \dots, R_n \rangle$ )
  global stop_nodes
   $\text{next\_step}_1, \dots, \text{next\_step}_n \leftarrow \emptyset$ 
  for all  $i \in \{1, \dots, n\}$  do
    last_step  $\leftarrow$  last element of  $R_i$ 
    visited_nodes  $\leftarrow \bigcup_{V \in R_i} V$ 
    for all  $v \in \text{last\_step} \setminus \text{stop\_nodes}$  do ▷ Collect valid neighbors of nodes in previous step
       $\text{next\_step}_i \leftarrow \text{next\_step}_i \cup v \bullet \cup \bullet v \setminus \text{visited\_nodes} \setminus \text{stop\_nodes}$ 
    end for
    if  $\text{next\_step}_i = \emptyset$  then
      fail ▷ If a module can't expand, fail
    end if
  end for
  return  $\langle \text{next\_step}_1, \dots, \text{next\_step}_n \rangle$ 
end procedure

```

In every even step, the shells contain only place vertices, which have to be disjoint between modules, as stated in Lemma 1. The simplest method to handle overlapping places is to mark them as forbidden and remove them from the expansion step, as seen in Alg. 3. This is a point of concern for improving the quality of the resulting modularization, since it is likely to introduce small "strips" of places which run between the edges of modules and act as a buffer (see Fig. 2b). The buffer introduces synchronization points, which can reduce the effectiveness of the state space reduction achieved by the modular approach.

Algorithm 3 Subroutine to find and resolve conflicts between expansion steps

```

procedure RESOLVECONFLICTS( $\langle R_1, \dots, R_n \rangle, \langle \text{step}_1, \dots, \text{step}_n \rangle$ )
  global stop_nodes
   $V_i \leftarrow \bigcup_{R \in R_i} R, \forall i \in \{1, \dots, n\}$   $\triangleright V_i$  contains all nodes from previous steps
  for all  $i \in \{1, \dots, n\}$  do
    for all  $v \in \text{step}_i$  do
      if  $v \in (\text{step}_1 \cup \dots \cup \text{step}_n) \setminus \text{step}_i$  or  $v \in (V_1 \cup \dots \cup V_n) \setminus V_i$  then
        conflicts  $\leftarrow$  conflicts  $\cup \{v\}$ 
        stop_nodes  $\leftarrow$  stop_nodes  $\cup \{v\}$ 
      end if
    end for
    if  $|R_i| \bmod 2 = 0$  then  $\triangleright$  Even number of previous steps  $\hat{=}$  current step contains places
       $\text{step}_i \leftarrow \text{step}_i \setminus \text{conflicts}$   $\triangleright$  Future work: find partner in conflicting modules
    end if
  end for
  return  $\langle \text{step}_1, \dots, \text{step}_n \rangle$ 
end procedure

```

3.2.3. Equalizing the step sizes

The next reduction of the expansion step concerns itself with the size of the individual shells. Since the process attempts to maintain isomorphic substructures after each expansion, an equal number of vertices has to be added to each module in a given step. If the step sizes differ, all candidate shells have to be reduced to the size of the smallest one. This procedure is shown in Alg. 4. When deconstructing a Petri net system into modules, the inclusion of a place $p \in P$ in a module requires the inclusion of all transitions $t \in T : (p, t) \in F \cup F^{-1}$ in the same module. Thus, if the current expansion steps consist of transitions, but differ in size, it is impossible to remove transitions from a candidate shell, and this reduction of the steps has to fail.

However, the inverse is not required, which allows us to remove places from candidate shells of expansion steps with an even number. One method of constructing equally sized subsets of shells with a higher likelihood of resulting in an isomorphic graph utilizes an idea similar to the signature s , which we previously calculated for selecting the seed vertices: Since we are only concerned with the immediate next expansion step, we calculate a modified signature $s'_k(v)(i, o) = |\{c(v, v') = o, c(v', v) = i, v' \in V_k^{Visited}\}|$, where $V_k^{Visited}$ is the union of all previous expansion steps. This way, $s'_k(v) : (\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N})$ describes the connection structure of a node $v \in V_k^{Shell}$ to only its neighbors in the module k .

To preserve the isomorphism between modules, nodes v_1, \dots, v_n are only kept in the shells of their respective module k when they have the same number of incoming and outgoing edges to the previous shell of the module, so $s'_k(v_1) = \dots = s'_k(v_n)$. If there is the same number of vertices v_i with the same signature $s'_k(v_i)$, all of these vertices remain in the shell. If this number of vertices differs between modules, the shell with the smallest number determines how many vertices of the other shells are kept. The selection heuristic of these vertices is subject to further research.

After each reduction procedure, it is possible that a candidate expansion step is completely empty. Whether the step was empty from the beginning because all neighbors of the previous shell were marked as forbidden, were removed because they conflicted with other modules or emptied because their specialized signature s' didn't match any of the other candidates, the algorithm can't grow the modules in this step and therefore not in any future steps. In this case, the algorithm fails and returns the last valid modules.

Algorithm 4 Subroutine to equalize the size of expansion steps

```

procedure EQUALIZESTEPS( $\langle R_1, \dots, R_n \rangle, \langle \text{step}_1, \dots, \text{step}_n \rangle$ )
  next_step1, ..., next_stepn  $\leftarrow \emptyset$ 
  if  $|R_1| \bmod 2 = 1$  then  $\triangleright$  Odd number of previous steps  $\hat{=}$  current step contains transitions
    fail  $\triangleright$  Transitions can't be excluded from expansion
  end if
  for all  $v \in \text{step}_1 \cup \dots \cup \text{step}_n$  do
    signaturev  $\leftarrow ((i, o) \mapsto |\{v' \mid W(v', v) = i \wedge W(v, v') = o\}|)$ 
  end for
  signatures =  $\bigcup_{v \in \text{step}_1 \cup \dots \cup \text{step}_n} \text{signature}_v$ 
  for all  $s : (\mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}) \in \text{signatures}$  do
     $s_i \leftarrow \{v \in \text{step}_i \mid \text{signature}_v = s\}, \forall i \in \{1, \dots, n\}$ 
    if  $|s_1| = \dots = |s_n|$  then
      next_stepi  $\leftarrow \text{next\_step}_i \cup s_i, \forall i \in \{1, \dots, n\}$ 
    else
      next_stepi  $\leftarrow \text{next\_step}_i \cup \{\text{choose MIN}\{|s_1|, \dots, |s_n|\} \text{ nodes from } s_i\}$   $\triangleright$  Future
      Work
    end if
  end for
  return  $\langle \text{next\_step}_1, \dots, \text{next\_step}_n \rangle$ 
end procedure

```

3.2.4. Calculating isomorphisms

The last step after performing the aforementioned reductions is to verify that the calculated expansion step maintains the isomorphism between the modules. The module graphs are obtained by constructing the graphs induced from the net graph by all vertices contained in all steps of each module. Due to ISOMORPHISM being a problem in NP, this part consumes most of the runtime and is subject to several heuristic approaches to reduce the complexity. Since the isomorphism between the modules is transitive, it is sufficient to find an isomorphism σ_i between each replica R_i and R_1 , and not necessary to check isomorphism between every pair of modules, which is shown in Alg. 5.

If an isomorphism check fails, the expansion step can't be performed and the algorithm fails, again returning the last previously valid modules. Otherwise, the expansion steps are added to their respective modules in preparation for the next iteration of the loop.

Algorithm 5 Subroutine to check the isomorphism between modules

```

procedure CALCULATEISOMORPHISM( $\langle R_1, \dots, R_n \rangle, \langle \text{step}_1, \dots, \text{step}_n \rangle$ )
   $G_i \leftarrow \text{INDUCEDSUBGRAPH}(N, \text{step}_i \cup \bigcup_{R \in R_i} R)$ 
  for all  $i \in \{2, \dots, n\}$  do
     $\sigma_{i-1} \leftarrow \text{ISOMORPHISM}(G_1, G_i)$ 
    if  $\sigma_{i-1} = \emptyset$  then
      fail
    end if
  end for
  return  $\langle \sigma_1, \dots, \sigma_{n-1} \rangle$ 
end procedure

```

3.2.5. Post-processing and improvements

The result of the aforementioned algorithm contains the identified replications R_1, \dots, R_n , each represented by a sequence of expansion steps, as well as the isomorphic mappings $\sigma_1, \dots, \sigma_{n-1}$, so that σ_1 is the mapping between R_1 and R_2 . We can collect all places from a replication representation R_i as $P_i = \bigcup_{R \in R_i} (R \cap P)$, where P is the set of places of the input Petri net system. The set of all places not included in the replicated modules is $P_{n+1} = P \setminus \bigcup_{i \in \{1, \dots, n\}} P_i$. Since $\{P_1, \dots, P_n, P_{n+1}\}$ is a partition of P , it induces a modular structure $\mathcal{M}' = [\mathcal{I}', \mathcal{F}']$, with the set of instances $\mathcal{I}' = \{[N'_1, m'_{01}], \dots, [N'_{n+1}, m'_{0(n+1)}]\}$ according to lemma 1. To combine these replicated modules into replicated instances of a single module, we define a modular structure $\mathcal{M} = [\mathcal{I}, \mathcal{F}]$ as follows:

- $\mathcal{I} = \{[N_{repl}, j, m_{0j}] \mid j \in \{1, \dots, n\}\} \cup \{[N_{rest}, n+1, m_{0(n+1)}]\}$,
where $m_{0j}(p) = m(\sigma_{j-1}(p))$ for $j \in \{1, \dots, n\}$ and $m_{0(n+1)}(p, n+1) = m(p)$.
- \mathcal{F} is calculated analogously to Lemma 1 combined with \mathcal{I} .

Due to the nature of the expansion, the replicated module N_{repl} is connected, while the rest module N_{rest} may not be. Therefore, N_{rest} can w.l.o.g be split into connected modules, which can further improve the state space reduction.

The aforementioned algorithm can be improved in different areas by making assumptions about the isomorphic mappings between the modules. While the seed vertices are selected in a way that incentivizes them to be mapped onto each other early in the process, it is entirely possible for that to change after a few successful expansion steps. However, after some number r of steps, we can observe that the isomorphic mapping only frequently changes for some ε outermost shells of the modules, while the mapping is stable in shells which are closer to the center.

With this assumption, we can reduce the size of the input graphs for the isomorphism problem. The more successful expansion steps are performed, the larger the graphs, but the more stable the center. Let S_k^i be the set of nodes included in the i -th shell of the k -th module, and r be the index of the currently planned expansion step, with all necessary reductions already performed. Furthermore, let σ_k be the isomorphic mapping from module k to $k+1$ after step $r-1$.

Previously, the graphs G_k were induced by the set of vertices $\bigcup_{0 \leq i \leq r} S_k^i$. Under the aforementioned assumption, the size of this set can be reduced by only including vertices from an "inner" shell if they're connected to a vertex in an "outer" shell. When calculating the new isomorphic mapping σ'_k , we introduce the additional constraint that $\sigma'_k(v) = v'$ iff $\sigma_k(v) = v' \wedge v \in \bigcup_{1 \leq i \leq r-\varepsilon} S_k^i$, so inner vertices are always mapped to other inner vertices without having to include all other inner vertices.

Another application of this assumption can be found during the step size reductions. When a place p is included in the candidate expansion steps of multiple modules, the simplest resolution is to include this place in neither module. However, as described before, this reduces the result quality by introducing strips of places along the module borders. To counteract this behavior, it would be preferable to allocate the conflict place to a module instead of excluding it entirely. Since the number of vertices has to be equal across the planned expansion steps, it is necessary to find places that can be included into the other steps without violating the isomorphism. Under the assumption that the isomorphism tends to be stable after some steps, the isomorphic mapping from the previous step can be utilized to identify those places. Further research and experiments are required to validate the effectiveness of this approach by examining the success rate of such an allocation.

The last area where the previous isomorphic mappings can be used is during the equalization of the step size. When the expansion steps contain a differing number of places with the same signature, identically sized subsets of those places have to be identified. The previous isomorphism could be utilized similarly as during the conflict resolution described before: By extending the modified signature s'_k to include the isomorphism, we reduce the flexibility of changing the mapping of recently included vertices, but in turn gain a more sound heuristic that can reduce backtracking or unnecessary failures.

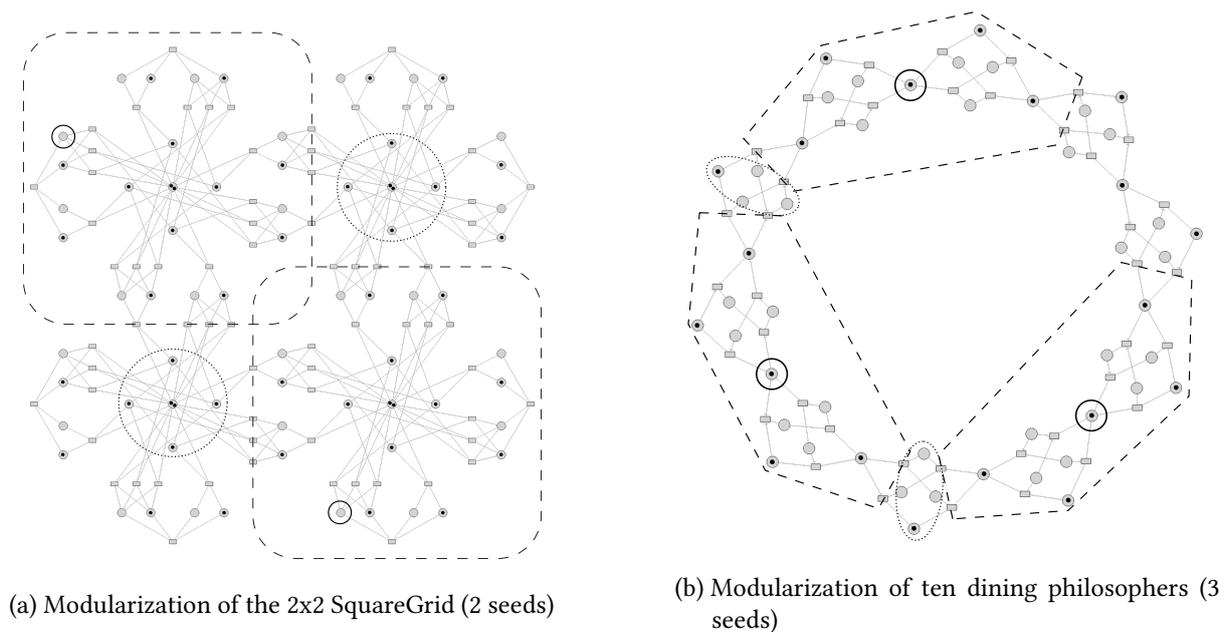


Figure 2: Results of the FINDREPLICATIONS procedure

3.2.6. Implementation

The prototype implementation is written in Python for its development flexibility and library support. It utilizes the widely known `networkx` library [9] to manage the graph data structures and, more importantly, check and calculate the isomorphisms between modules using an implementation of the VF2 algorithm [10]. Our presented algorithm performs the isomorphism checks in a loop, and while the heuristic described in the previous section can take some strain off of the calculations by reducing the size of the input graphs, there’s more performance to be gained. It is likely that a specialized implementation of the isomorphism check could exploit the iterative nature of its execution to reduce redundant calculations, which currently have to be performed anew in every call.

The current implementation supports the selection of seed vertices via their signature, as well as basic strategies for conflict resolution and step size equalization. Because of its impact on the runtime, the isomorphism calculation accepts several parameters to control the size of the stable area in the center, which reduces the graph size. Despite lacking the more sophisticated strategies that utilize isomorphic mappings from previous iterations to allocate conflicting places to modules and improve the result quality during step size equalization, the implementation finds sufficiently large replicated modules on several models.

Figure 2a shows the result of the algorithm on the `SQUAREGRID` model [11] with the two seed vertices encircled in the top left and bottom right. After five successful expansion steps, the two modules meet on the center places of their adjacent grid cells, highlighted by the dashed circle. At this point, the algorithm can’t expand further due to the missing ability to allocate places between modules, causing it to terminate. An allocation strategy would be able to resolve this conflict by allowing the top left module to expand to the right, but not downwards, leaving those places for the bottom right module. The same kind of results can be observed on larger versions of the model, especially when the expansion is performed from more than two seed places.

The results of the automatic modularization of another model is shown in Figure 2b. This is a version of the well-known dining philosophers model with ten philosophers. Vertices in this model only have one of two different signatures, where one signature is shared only by the ten forks while the other is common to the remaining 40 places. This makes the forks much more desirable starting points. As in Figure 2a, modules are in conflict on places between each other, and again those conflicts could be resolved by the aforementioned method. While it is impossible to make out from the figure alone,

the philosopher to the right of the top module actually grabs his forks in the opposite order from his colleagues, which prevents the application of state space reduction methods using symmetries, but leaves this approach unfazed since the philosophers fully contained in the replications still share the same behavior.

All in all, the current state of the implementation performs well in terms of result quality, especially for models which contain highly replicated structures. For models where the desired modules are less obvious to the naked eye, the prototype often terminates early due to its lacking conflict resolution abilities. Since the main goal of the prototype was originally to provide a framework for easy-to-implement validation of modularization ideas, the runtime performance was not a large concern during early stages of development. However, the described assumptions about the stability of isomorphic mappings between the centers of modules resulted in significant performance gains, especially for models with high connectivity, and we expect another boost from utilizing an iteration-optimized version of the isomorphism checking algorithm.

4. Non-Replicated Modularization

Modularization based on replications improves the analysis of Petri net systems by reducing redundant state space. In this chapter we will highlight other methods in order to deal with situations where we cannot expect to find replications. For example, if all replications have already been found in a given Petri net system, the rest of the network may still be large. In this case, we then could try to modularize the rest of the network with other methods. Another case would be the reduction of already found replications. The previously described approach always tries to find the largest possible replications. The size of the replications is not limited upwards, as long as the propagation does not collide with other replications. The replications can therefore become rather big. To leverage the advantage of modularization even further, it is possible to continue modularizing a replica again using other methods. The additional modularization can then be transferred from the replica to the other without any additional effort. Although Definition 5 encourages us to distinguish between the concepts of module and instance, we decline to do so in this section. Since we assume here that we will not find any replicas, i.e. instances, that belong to the same module, the association to the structure of modules is not required. From now on we will only use the term instances as components of a modular Petri net. The goal remains the modularization of a given Petri net system, as described in Definition 6.

With respect to the application cases described above, we do not mind whether the given Petri net system itself is a just created instance, the remaining Petri net system or a completely new net; in any case we can build a modular Petri net out the given net. The following approaches aim to generate a partitioning of the place set of a Petri net, which can then be converted into a modular structure as described in lemma 1.

4.1. T-Invariants

Decompositional analysis based on the structure of a Petri net system is obvious because of the strong connection between structure and behavior of Petri net systems. In [12] they advance the analysis by adding hierarchy through the nested unit Petri nets model. The Problem of dividing a Petri net system into functional subnets, a net with a denoted set of input and output places, is discussed in [13]. Invariants for the modularization of Petri net systems have been used before; [14] used place invariants to abstract a set of places to one place. This abstracted Petri net system tends to be simpler to analyze. Our approach here is to use transition invariants to generate instances with significant internal behavior. Transition invariants are a criterion for cyclic, thus non-trivial behavior. A cycle in the state space responds to a transition invariant of the according Petri net system [15]. Note, that the existence of a transition invariant is only a necessary criterion for the existence of cyclic behavior; we may find a transition invariant we can never execute as it is not marked in any reachable marking. The use of t-invariants has already been used for the decomposition of Petri net systems due to their importance in the state space. In [1], a Petri net system is clustered on the basis of t-invariants, so that

resulting clusters correspond to functional units in biological networks. Here, however, the focus was on the clustering of similar invariants.

Definition 10 (Incidence Matrix). A Petri net system $N = [P, T, F, W, m_0]$ can be represented as an *incidence matrix* $C^{|P| \times |T|}$ where $c_{ij} = W((t_j, s_i)) - W((s_i, t_j))$ for $i \in \{1, \dots, |S|\}$ and $j \in \{1, \dots, |T|\}$.

Definition 11 (Transition Invariants, Support). Let N be a Petri net system and C its incidence matrix. A *transition invariant* (*t-invariant*) $\vec{i} \in \mathbb{Z}^{|T|}$ is a solution of the homogenous linear system of equations $C \cdot \vec{i} = \vec{0}$. The *support* $\text{supp}(\vec{i}) \subseteq T$ of the transition invariant \vec{i} is the set of transitions whose entry is not zero in \vec{i} .

Informally, a t-invariant describes a multiset of transitions whose execution will not change the state of the system. After firing all transitions, the net will be situated in the very same marking as before. We only consider non-trivial, non-negative t-invariants. The trivial t-invariant $\vec{0}^{|T|}$ provides no useful information for us. A negative entry in a t-invariant would represent a reverse firing of a transition which is not applicable. Despite this restriction, we still can get infinitely many t-invariants. Therefore, we focus on minimal invariants.

Definition 12 (Minimal Transition Invariants). For a *minimal* transition invariant $\vec{i} \in \mathbb{Z}^{|T|}$, it exists no other t-invariant $\vec{i}' \in \mathbb{Z}^{|T|}$ with $\text{supp}(\vec{i}') \subset \text{supp}(\vec{i})$ for $\vec{i}' \neq \vec{i}$. Also, the least common divisor of the entries of \vec{i} is 1.

From now on, if we mention t-invariants, we always refer to minimal non-trivial, non-negative t-invariants, as we only consider those kinds of t-invariants.

T-invariants serve as the bases for prospective instances of the modular Petri net. Instances should be disjoint per definition, thus we are only interested in t-invariants that have a disjoint set of support transitions. The requirement to compute disjoint t-invariants can be easily implemented in an iterative computation by claiming the weights of already computed support transitions to be zero in future invariants. Thus, they do not occur in support of further t-invariants. Even for disjoint t-invariants, the preplaces resp. postplaces of the support transitions may overlap. In accordance with the definition of instances in a modular structure, the sets of places need to be disjoint as well. To resolve this problem, we introduce the concept of super-disjointness.

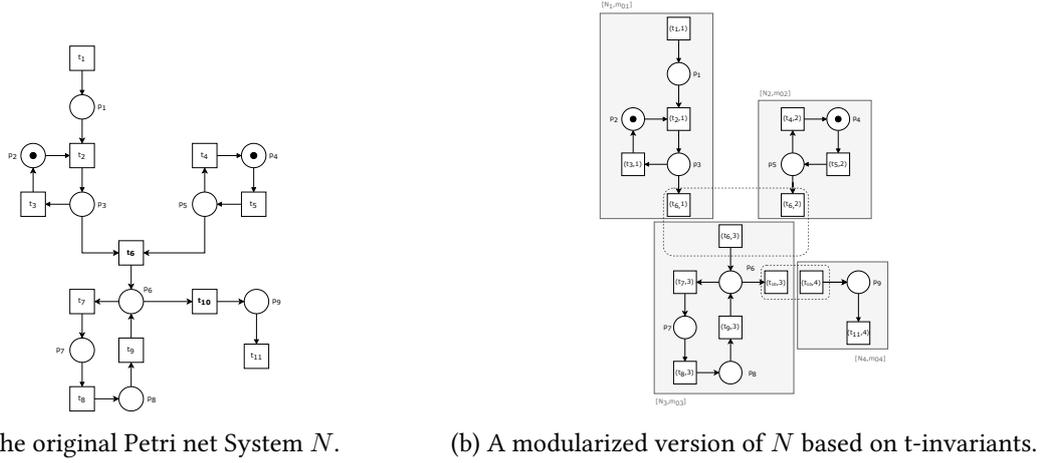
Definition 13 (Super-Disjointness). Let N be a Petri net system and C its incidence matrix. Two t-invariants $\vec{i}, \vec{i}' \in \mathbb{Z}^{|T|}$ are *super-disjoint*, if

1. they are disjoint, i.e. $\text{supp}(\vec{i}) \cap \text{supp}(\vec{i}') = \emptyset$
2. the environments of the support transitions are disjoint as well; i.e. $\bigcup_{t \in \text{supp}(\vec{i})} (\bullet t \cup t \bullet) \cap \bigcup_{t' \in \text{supp}(\vec{i}')} (\bullet t' \cup t' \bullet) = \emptyset$

For a Petri net system, super-disjoint t-invariants can be calculated with a tool for Petri net system analysis, for example LoLA [16]. Calculating positive t-invariants is closely related to the integer linear programming problem, which is known to be NP-complete, therefore calculating t-invariants is not a trivial problem. Due to the restriction of super-disjointness, the calculation of will be executed $|T|$ times worst case.

We now assume we have generated a set of super-disjoint t-invariants $\{\vec{i}_1, \dots, \vec{i}_\ell\}$. From every invariant \vec{i}_j , we generate a set of places $P_j = \bigcup_{t \in \text{supp}(\vec{i}_j)} \bullet t \cup t \bullet$ for $j \in \{1, \dots, \ell\}$. With reference to Lemma 1, we can generate a modular structure out of the set of created place sets $\{P_1, \dots, P_\ell\}$.

Example 3 (super-disjoint T-Invariant Modularization). Let N be a Petri net system, as depicted in Figure 3a We can calculate three super-disjoint t-invariants $\vec{i}_1, \vec{i}_2, \vec{i}_3$, for which we can identify the support: $\text{supp}(\vec{i}_1) = \{t_1, t_2, t_3\}$, $\text{supp}(\vec{i}_2) = \{t_4, t_5\}$ and $\text{supp}(\vec{i}_3) = \{t_7, t_8, t_9\}$. As described we generate the place set of the future instances from the environment of the support transitions. This results in $P_1 =$

(a) The original Petri net System N .(b) A modularized version of N based on t-invariants.**Figure 3:** Exemplary representation of the modularization of Petri net system N .

$\{p_1, p_2, p_3\}$, $P_2 = \{p_4, p_5\}$ and $P_3 = \{p_6, p_7, p_8\}$. The only place p_9 of the rest net determines another instance with $P_4 = \{p_9\}$. According to Lemma 1, we can generate the modular structure. Firstly, we generate the transition sets corresponding to the four place sets: $T_1 = \{(t_1, 1), (t_2, 1), (t_3, 1), (t_6, 1)\}$, $T_2 = \{(t_4, 2), (t_5, 2), (t_6, 2)\}$, $T_3 = \{(t_6, 3), (t_7, 3), (t_8, 3), (t_9, 3), (t_{10}, 3)\}$ and $T_4 = \{(t_{10}, 4), (t_{11}, 4)\}$. Arcs and Weights and initial markings are defined appropriately. We can deduce that t_6 of N is a fusion transition and $(t_6, i) \in T_{i|interface}$ for $i \in \{1, 2, 3\}$. The same applies for t_{10} and (t_{10}, i) for $i \in \{3, 4\}$. All other transitions of the instances are internal ones. This leads to two fusion vectors $\mathcal{F} = \{[(t_6, 1), (t_6, 2), (t_6, 3)], [(t_{10}, 3), (t_{10}, 4)]\}$. The modular Petri net generated out of the modular structure is depicted in Figure 3b.

It is reasonable to assume instances generated on t-invariants do not modularize the Petri net system completely. Parts of the original Petri net system may remain unaffected by the modularization and form one instance by itself. That these parts are connected is improbable, therefore the analysis of this rest-instance would require a lot of synchronization with other instances. This is not consistent with the concept of individual analysis of the instances, which will accelerate the modular state space analysis. Furthermore, we assume that t-invariants tend to generate small instances with little internal behavior. It is therefore convenient to combine partial instances under certain conditions to form a larger instance. The most obvious approach is to join two instances that share a fusion set. This can be accomplished by unifying the place sets of the two instances and forming a new instance according to Lemma 1. This is possible without restriction, since the place sets are disjoint by construction. It is also possible to combine instances that are close to each other. For this purpose, we can calculate the shortest path between two instances in pairwise order. The goal is to merge the instances and to include the nodes on this shortest path into the new instance. In order to realize this, we extract the places that are located on the shortest path and join them with the union of the place sets of the instances. The whole set then serves as the basis for a new, larger instance. In this way, residual parts of the Petri net system, that lie between two instances are to be included in one instance. Doing so creates connected instances, which tend to have more internal behavior. It also reduces the unmodularized part of the Petri net system. The instances should not become too large, of course. It is recommended to define a maximum size for instances as the analysis of (too) large instances can reduce the efficiency increase achieved by modularization. The exploitation of t-invariants provides a legitimate way to approach the modularization of a Petri net system. The computational effort seems comparatively lightweight, so with the proposed extensions we see this method as an extensible enrichment in modularization.

4.2. Minimal Interfaces

Another legitimate goal for modularization is the generation of the smallest possible interfaces. For the analysis of the state space of a modular Petri net, internal transitions and interface transitions of instances are treated differently. For the activation and firing of internal transitions only the state of the corresponding instance is relevant. The firing behavior of interface transitions depends on the states of all attached instances. The analysis must therefore include all instances affected. However, the efficiency increase of the analysis of modular state spaces follows from restricting the analysis to single instances. It is therefore a valid modularization criterion to keep the number of interface transitions as low as possible and thus to reduce behavior that affects several instances. For more details on the analysis of the modular state space and the structures required for it, we reference [3]. This subsection provides two approaches to generate a modular Petri net system with little interface behavior.

4.2.1. Modularization by Hypergraph Partitioning

In [17], a system described only as an incidence matrix between states and actions is also to be modularized to speed up the analysis. Here, the data flow between the resulting modules should be minimized, i.e. the interface actions should be kept as small as possible. Finding minimal data flow between modules refers to the problem of finding a minimal cut.

The method for this is to partition the matrix using hypergraph partitioning.

Definition 14 (Hypergraph [18]). A *hypergraph* $H = [V^H, E^H]$ is defined as a set of vertices V^H and a set of hyperedges E^H between the vertices. A hyperedge $e \in E^H$ is a subset of vertices: $e \subseteq V^H$.

Thus, a graph is a special instance of a hypergraph, where every hyperedge connect exactly two vertices.

Definition 15 (Hypergraph Partitioning [18]). Let $H = [V^H, E^H]$ be a hypergraph. The set of vertex sets $\Pi = \{V_1^H, V_2^H, \dots, V_\ell^H\}$ is an ℓ -*partition* of H , if the following conditions hold:

- each set V_i^H is a nonempty subset of V^H , i.e. $\emptyset \neq V_i^H \subseteq V$ for $i \in \{1, \dots, \ell\}$
- the sets are pairwise disjoint, i.e. $V_i^H \cap V_j^H = \emptyset$ for $i \neq j \in \{1, \dots, \ell\}$
- the union of all sets results in V^H , i.e. $\bigcup_{i \in \{1, \dots, \ell\}} V_i^H = V^H$

We refer to $E_{extern}^H = \{e \in E^H \mid \exists v_i \neq v_j \in e \exists V_i^H \neq V_j^H \in \Pi : v_i \in V_i^H \wedge v_j \in V_j^H\}$ as the *external hyperedges*.

A hypergraph partitioning divides a hypergraph into a set of disjoint subsets $V_1^H, V_2^H, \dots, V_\ell^H$. The external hyperedges connect vertices from distinct partitions. In [17], they reduce the problem of generating as little data flow as possible to hypergraph ℓ -partitioning with minimal cost for $\ell > 0$. The cost of a partitioning is related to the number of external hyperedges in the following way:

Definition 16 (Cost of a Hypergraph Partitioning). Given a hypergraph $H = [V^H, E^H]$, the *cost* of an ℓ -partitioning $\Pi = \{V_1^H, V_2^H, \dots, V_\ell^H\}$ is defined as $\lambda(\Pi) = |E_{extern}^H|$.

In [18], they additionally present a balance criterion.

Definition 17 (Balanced Partition, Balance Criterion [18]). Let $H = [V^H, E^H]$ be a hypergraph and $\Pi = \{V_1^H, V_2^H, \dots, V_\ell^H\}$ an ℓ -partitioning of H . For every vertex $v \in V^H$, $w(v)$ denotes a weight of v . For every partition V_i^H for $i \in \{1, \dots, \ell\}$, we define $w(V_i^H) = \sum_{v \in V_i^H} w(v)$ as the weight of V_i^H . Over all partitions, we can calculate the average weight of the partitions: $w_{avg} = \sum_{i=1}^k w(V_i^H)/k$. Partition Π is said to be *balanced* if it fulfills the *balance criterion*: $w(V_i^H) \leq w_{avg}(1 + \varepsilon)$.

Here, ε denoted a predefined maximum imbalance ratio allowed. The bigger we set ε , the more the partitions are permitted to differ. The hypergraph partitioning problem is defined as follows:

HYPERGRAPH ℓ , PARTITIONING**Input:** Hypergraph H , $\ell \in \mathbb{Z}^+$ **Question:** Find a balanced ℓ -partitioning $\Pi = \{V_1^H, V_2^H, \dots, V_\ell^H\}$, such that $\lambda(\Pi)$ is minimal.

This problem is NP-hard [19]. Therefore, we do not expect to find a hypergraph partitioning with the minimal cost; however, a heuristic approximate solution satisfies our requirements.

The following step is an interpretation of a Petri net system as a hypergraph, for which we then aim to generate a balanced partitioning with minimal cost. Therefore, we introduce the incidence hypergraph.

Definition 18 (Incidence Hypergraph). Given a Petri net system $N = [P, T, F, W, m_0]$, the *incidence hypergraph* $I = [V^I, E^I]$ is an undirected, bipartite graph, where

- $V^I = P$
- $E^I = \text{bags}(\bullet t \cup t \bullet \mid t \in T)$ (with $\text{bags}(X)$ being a multiset with elements from X .)

Note, that in contrast of the Definition 14, the set of edges is defined as a multiset of subsets of places. The reason is, that for a Petri net system, the hyperedges should represent the transitions. The conventional hypergraph definition would compress transitions with the same set of pre/postplaces to one hyperedge, so structural information about the Petri net system would be neglected. For example, consider Petri net system N in Figure 4a. If we add a transition t_0 with $\bullet t_0 = \{p_2\}$ and $t_0 \bullet = \{p_1\}$, this would result in a hyperedge $\{p_1, p_2\}$, which is already part of E^I representing transition t_1 . Consequentially, no additional edge would be introduced and t_0 becomes invisible. To avoid this, we adapt the definition for the use case of Petri net systems. As the weights are not important, the incidence hypergraph abstracts this information. For incidence hypergraph I we aim to generate a balanced ℓ -partition $\Pi = \{V_1^I, V_2^I, \dots, V_\ell^I\}$ with minimal cost $\lambda(\Pi)$. In our use case, we can weaken the balancing criterion formulated in Definition 17. As for now, the places of a Petri net system that form the vertices of the incidence hypergraph are all equally valid, we can neglect the weight of the vertices. To us, it is sufficient to have partitions with approximately the same size.

Definition 19 (Balance Criterion for Incidence Hypergraph Partitioning). Let $I = [V^I, E^I]$ be an incidence hypergraph of a Petri net system N and $\Pi = \{V_1^I, V_2^I, \dots, V_\ell^I\}$ an ℓ -partitioning of H . Partitioning Π is balanced, if every partition V_i^I for $i \in \{1, \dots, \ell\}$ fulfills the following *balance criterion*: $|V_i^I| \leq |V^I|/\ell \cdot (1 + \varepsilon)$.

Each partition must not differ in size by more than ε -fold from the average size of a partition. Solving the hypergraph partitioning problem for the incidence hypergraph of a Petri net system will result in a partitioning of places and an identification of transitions as internal and interface. This information is sufficient to build a modular Petri net. Partition Π divides the vertices of the incidence hypergraph I into partitions $\{V_1^I, V_2^I, \dots, V_\ell^I\}$. If we take this down to the Petri net system level, the partitions correspond place sets $\{P_1, P_2, \dots, P_\ell\}$, such that $P_j = V_j^I$ for $j \in \{1, \dots, \ell\}$. Each place set P_j then forms the place set of an instance $[N_j, m_{0j}]$ for $j \in \{1, \dots, \ell\}$. In accordance to Lemma 1 and with reference to our initially given Petri net system N , we can generate a modular structure, thus a modular Petri net out of $\{P_1, P_2, \dots, P_\ell\}$. Due to the balance criterion, the instances are approximately the same size while keeping the number of fusion transitions as small as possible.

Example 4. Let N be a given Petri net system, depicted in Figure 4a. For N , we generate the incidence hypergraph I , depicted in Figure 4b. A balanced 2-partitioning for I would be $\Pi = \{\{p_1, p_2, p_3\}, \{p_4, p_5, p_6\}\}$ with only one external hyperedge $E_{extern}^I = \{\{p_3, p_4, p_5\}\}$. This partitioning is balanced, as the partitions are the same size, and minimal - less than one hyperedge would not be possible for a connected underlying Petri net system. A resulting modular structure would contain two instances $[N_1, m_{01}], [N_2, m_{02}]$, where $P_1 = \{p_1, p_2, p_3\}, T_1 = \{(t_1, 1), (t_2, 1), (t_3, 1)\}$ and $P_2 = \{p_4, p_5, p_6\}$ and $T_2 = \{(t_3, 2), (t_4, 2), (t_5, 2)\}$. The other components are defined appropriately. The only fusion set would be $f = \{(t_3, 1), (t_3, 2)\}$.

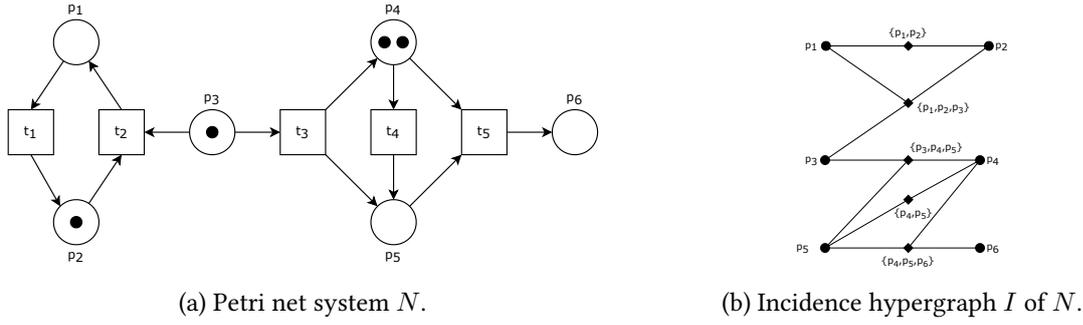


Figure 4: Incidence hypergraph of a given net

For solving the hypergraph partitioning problem, there exist several tools, like [18] or [20]. In [21], they provide a parallel approach to handle the severity of the problem. The problem remains NP-hard, which is why the approaches are only heuristic. In our case, the application of hypergraph partitioning may not necessarily deliver optimal results, nevertheless, in the following subsection we want to consider an approach which solves another applicable problem in polynomial time.

4.2.2. Modularization by Dual Hypergraph Cutting

In this chapter, we highlight a method for modularizing Petri net systems using the dual incidence hypergraph. For this purpose, further graph-theoretic definitions are required.

Definition 20 (Dual Hypergraph [22]). Let $H = [V^H, E^H]$ be a hypergraph. The *dual hypergraph* $H^* = [V^{H^*}, E^{H^*}]$ is a hypergraph, where $V^{H^*} = E^H$ and $E^{H^*} = \{\{e \mid e \in E^H \wedge v_1 \in e\}, \{e \mid e \in E^H \wedge v_2 \in e\}, \dots, \{e \mid e \in E^H \wedge v_n \in e\}\}$ for $\{v_1, v_2, \dots, v_n\} = V^H$.

The edges of the hypergraph form the vertices of the dual hypergraph. For every vertex of a hypergraph, the set of hyper edges that contain this vertex corresponds to one edge of the dual hypergraph.

The hypergraph partitioning problem from Section 4.2.1 seeks interdisjoint sets of vertices connected by as few hyperedges as possible. The equivalent problem in the dual hypergraph now aims to find the smallest possible set of vertices to remove, such that the graph is no longer connected without this set [22]. Such a set of vertices is called a minimal cut in graph theory. To approach this concept, we first define the connectivity of a graph.

Definition 21 (k -Connectivity, Vertex Connectivity [23]). Let $k \in \mathbb{N}$. A Graph $G = [V, E]$ is k -connected, if $|V| > k$ and every subgraph $G - X$ for $X \subseteq V$ with $|X| < k$ is connected. The maximum k for that G is k -connected, we call the *vertex connectivity* $\kappa(G)$.

In other words, the $\kappa(G)$ defines a number of vertices we need to remove to destroy the connectivity of a graph. We can describe the set of vertices we need to remove more precisely as the minimal cut in a graph.

Definition 22 (Minimal Cut). For a Graph $G = [V, E]$, a subset $X \subseteq V$ is a *minimal cut*, if $G - X$ is not connected and $|X| = \kappa(G)$.

By removing a minimal cut, the connectivity of the graph is broken, and the graph decomposes into components that are not connected to each other. A minimal cut of a hypergraph can be found by computing a minimal cut of its 2-section. Any minimal cut in H is also a minimum cut in $[H]_2$ and vice versa.

Definition 23. 2-Section [22] Let $H = [V^H, E^H]$ be a hypergraph. The *2-section* of H is an undirected graph $[H]_2 = (V^H, E_2)$ where $e \in E^H : \{x, y\} \subset e \Rightarrow \{x, y\} \in E_2$.

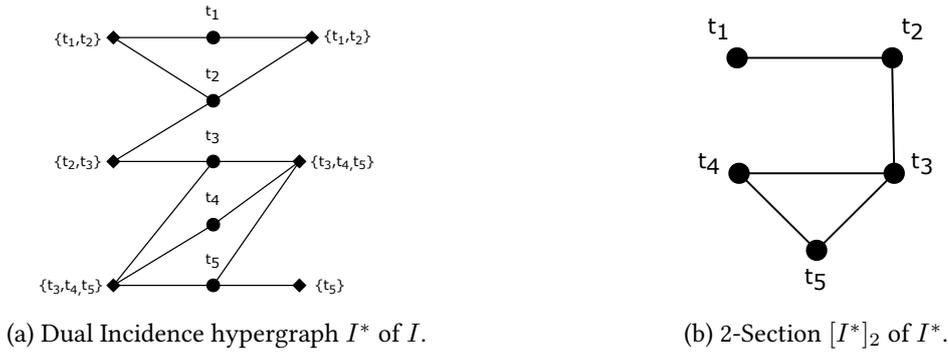


Figure 5: Dual Incidence hypergraph and 2-section

Thus, the 2-section of a hypergraph H contains all vertices of H and connects two vertices, if there is at least one hyperedge that contains those two vertices. In other words, the 2-section represents which vertices of H are connected to each other. As described in [24], a minimal cut can be found in polynomial time.

To use this knowledge to generate instances for a modular Petri net, we first generate the dual incidence hypergraph $I^* = [V^{I^*}, E^{I^*}]$ to incidence hypergraph I for a given Petri net system N , as defined in Definition 20. In I^* the vertices correspond to the transitions of N ; an edge represents a set of adjacent transitions for a particular place. Second, we create the 2-section of I^* as in Definition 23: $[I^*]_2 = [V^{I^*}, E_2^*]$, where $V^{I^*} = T$ and $\{t, t'\} \in E_2^*$, iff $(\bullet t \cup t \bullet) \cap (\bullet t' \cup t' \bullet) \neq \emptyset$. Thus, the 2-section $[I^*]_2$ contains all transitions of N as vertices. Two vertices are connected if the according transitions share a place in their environments. A minimal cut X of $[I^*]_2$ gives a set of transitions, we need to remove to disconnect the graph. This cut X leads to a set of external edges in the incidence hypergraph E_{extern}^I ; edges that lead from one partition of the incidence hypergraph to another partition of vertices. Vertices correspond to places of the underlying Petri net system, thus, we get a partitioning $\{P_1, \dots, P_\ell\}$ of P . With Lemma 1 we can deduce a modular structure which we can transform into a modular Petri net. Notice, that the generated partitioning is not necessarily balanced.

Example 5. For incidence hypergraph I in Figure 4b, we generate the dual incidence hypergraph I^* , where the vertices correspond to the transitions of N (cf. Figure 4a) and an edge represents a set of transitions that share one place in their environment. For example the edge $\{t_1, t_2\}$ expresses the fact, that t_1 and t_2 share one place, i.e. p_1 . They are connected with two distinct arcs, with which is evident from the fact that they also share two places, p_2 in addition to p_1 . The same holds for the transitions t_3, t_4 and t_5 . The 2-section of I^* is presented in Figure 5b. Here, the number of shared places is abstracted. Only the fact that two transitions are connected can be inferred from this representation.

The main issue with this approach is the lack of control over the number and size of instances. A minimal cut separates the net into at least two components, but possibly more. Furthermore, no formulation of a balancing criterion is possible, i.e. the size of the modules is not balanced; the focus here is only on finding a minimum cut. Cutting the dual incidence hypergraph leads to optimal, i.e. minimal sets of fusion transitions but possibly to the detriment of the size and number of modules.

5. Conclusion and Future Work

This paper provides approaches to automatically modularize Petri net systems. The main advantage in the modular state space analysis arises from the fact that several instances of the same module share the same state space if they have an analogous initial marking. Instances of the same module with different initial markings are still likely to have overlapping state spaces. These properties can be exploited during analysis to drastically reduce memory requirements. Therefore, the obvious approach to modularization is to first attempt to find replications. To incentivize that the replications contain

neither too little nor too much internal behavior, the `FINDREPLICATIONS` algorithm is controlled by a parameter which gives an upper bound for module size. If the identified replications are too small, the procedure can be restarted with a different set of seeds in an attempt to find another modularization with bigger modules. A prototype implementation shows the validity of the proposed method for models containing highly replicated structures, while yielding a starting point for improvements on less suitable models. Further improvements of the conflict handling employed by the process, of which several were suggested in 3.2, can push the point of termination further into the future, allowing the algorithm to find larger replicated modules.

After replications have been identified, further modularization methods can be applied. This can be used to further modularize both the remaining Petri net system and the replications themselves. If a replicated module is further modularized, the results can be propagated to its other instances via the isomorphic mappings, which are computed during the process. In addition to replication finding, three other methods were presented.

The first method, based on t -invariants, is rather difficult to compare with the other methods, since it was aimed at a different target. Here, the focus was on generating instances with nontrivial behavior. For this purpose, the concept of super-disjointness was introduced in order to derive disjoint instances from transition invariants. Those can be calculated efficiently. We also hope that the second method presented, based on hypergraph partitioning, will have an impact on module size, and thus behavior as well. Although this does not directly require the modules to have a certain size and structure, the balancing criterion expects the modules to be generated in a similar size. However, the main goal of this method was to minimize the interface behavior. This is achieved by finding a minimal ℓ -partitioning of the incidence hypergraph, as which the Petri nets system is represented for this purpose. The major drawback of this method is that the hypergraph partitioning problem is NP-hard and thus rather heuristically generated solutions can be expected. The third method overcomes this disadvantage and provides a way to efficiently generate minimal interfaces. The idea is to represent the Petri net system as a dual hypergraph and then determine a minimal cut. This leads directly to a partition of the places of the Petri net system and thus to a valid modularization. But here, too, we have to make concessions. With this approach, the instances generated can be of arbitrary size. The minimal cuts also may generate a variable number of instances. Based on the fact that the method of hypergraph partitioning allows a balancing criterion, which controls both - the module size and the interface size - we consider this method to be the most promising. Also supporting this is that hypergraph partitioning can be performed well in parallel.

Future Work might be an implementation of the other methods to compare their performance. The goal is to develop an implementation that offers different approaches to modularization. Those can be applied depending on the structure of the given Petri net system, resulting in the best possible modularization. This also includes further considerations of the structure. For example, net classes such as state machine or marked graph could be identified as distinct instances, making the analysis more efficient.

The quality of a modularization is not only a measure of how closely the modules match a modularization performed by a model designer. Instead, it is also measured by the amount of state space reduction gained during analysis of a modular Petri net compared to its unmodularized counterpart. To evaluate our presented methods further in that regard, tool support is required to generate the full modular state space. A quantitative way to measure result quality would open up a more in-depth way of finding generally suitable parameters, as well as exploring ways to calculate fitting parameters from the structure of the given Petri net model.

References

- [1] E. Grafahrend-Belau, F. Schreiber, M. Heiner, A. Sackmann, B. H. Junker, S. Grunwald, A. Speer, K. Winder, I. Koch, Modularization of biochemical networks based on classification of Petri net t -invariants, *BMC Bioinformatics* 9 (2008).

- [2] P. Buchholz, P. Kemper, Hierarchical Reachability Graph Generation for Petri Nets, *Formal Methods in System Design* 21 (2002) 281–315.
- [3] J. Gaede, S. Wallner, K. Wolf, Modular State Spaces - A New Perspective, 2024. Accepted at Petri Nets Conference 2024.
- [4] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, A Series of Books in the Mathematical Sciences, W. H. Freeman, New York, 1979.
- [5] M. Kuramochi, G. Karypis, Finding Frequent Patterns in a Large Sparse Graph, *Data Mining and Knowledge Discovery* 11 (2005) 243–271.
- [6] V. T. T. Duong, K. U. Khan, B.-S. Jeong, Y.-K. Lee, Top-k Frequent Induced Subgraph Mining Using Sampling, in: *Proceedings of the Sixth International Conference on Emerging Databases: Technologies, Applications, and Theory, EDB '16*, Association for Computing Machinery, New York, NY, USA, 2016, pp. 110–113.
- [7] A. Inokuchi, T. Washio, H. Motoda, An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data, in: D. A. Zighed, J. Komorowski, J. Żytkow (Eds.), *Principles of Data Mining and Knowledge Discovery*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2000, pp. 13–23.
- [8] E. Wong, B. Baur, S. Quader, C.-H. Huang, Biological Network Motif Detection: Principles and Practice, *Briefings in Bioinformatics* 13 (2012) 202–215.
- [9] A. A. Hagberg, D. A. Schult, P. J. Swart, Exploring network structure, dynamics, and function using networkx, in: G. Varoquaux, T. Vaught, J. Millman (Eds.), *Proceedings of the 7th Python in Science Conference*, 2008, pp. 11 – 15.
- [10] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub) graph isomorphism algorithm for matching large graphs, *IEEE transactions on pattern analysis and machine intelligence* 26 (2004) 1367–1372.
- [11] T. Shmeleva, D. Zaytcev, I. Zaytcev, Analysis of square communication grids via infinite petri nets (2009).
- [12] H. Garavel, Nested-unit petri nets: A structural means to increase efficiency and scalability of verification on elementary nets, in: R. R. Devillers, A. Valmari (Eds.), *Application and Theory of Petri Nets and Concurrency - 36th International Conference, PETRI NETS 2015*, Brussels, Belgium, June 21-26, 2015, *Proceedings*, volume 9115 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 179–199.
- [13] D. Zaitsev, Decomposition of petri nets, *Cybernetics and Systems Analysis* 40 (2004) 739–746.
- [14] C. Lakos, On the Abstraction of Coloured Petri Nets, in: *Application and Theory of Petri Nets 1997*, volume 1248, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 42–61.
- [15] K. Schmidt, Using Petri Net Invariants in State Space Construction, in: H. Garavel, J. Hatcliff (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2003.
- [16] K. Wolf, Petri Net Model Checking with LoLA 2, in: V. Khomenko, O. H. Roux (Eds.), *Application and Theory of Petri Nets and Concurrency*, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2018, pp. 351–362.
- [17] A. Ben Khaled El Feki, *Distributed Realtime Simulation of Numerical Models : Application to Power-Train* (2014).
- [18] Ü. Catalyürek, C. Aykanat, PaToH (Partitioning Tool for Hypergraphs), in: D. Padua (Ed.), *Encyclopedia of Parallel Computing*, Springer US, Boston, MA, 2011, pp. 1479–1487.
- [19] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Springer Science & Business Media, 2012.
- [20] G. Karypis, V. Kumar, Metis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices (1997).
- [21] A. Trifunovic, W. J. Knottenbelt, Parkway 2.0: A Parallel Multilevel Hypergraph Partitioning Tool, in: *International Symposium on Computer and Information Sciences*, Springer, 2004, pp. 789–800.
- [22] M. Dewar, D. Pike, J. Proos, Connectivity in Hypergraphs, *Canadian Mathematical Bulletin* 61 (2018) 252–271. Publisher: Cambridge University Press.
- [23] R. Diestel, *Graphentheorie*, 5 ed., Springer Spektrum, Berlin, 2017.

- [24] M. Stoer, F. Wagner, A Simple Min-Cut Algorithm, *Journal of the ACM (JACM)* 44 (1997) 585–591.