UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

COMPILAREA REȚELELOR PETRI: DE LA DESCRIERE FORMALĂ LA SIMULARE ȘI VIZUALIZARE

Absolvent

Mocanu Ștefan

Coordonator științific

Lect.dr. Claudia-Elena Chiriță

București, iunie 2021

Rezumat

Acest proiect propune un instrument de tip linie de comandă pentru parsarea și simularea Petri Neturilor, cu suport pentru extensii precum tokenuri colorate, arce inhibitoare, constrângeri temporale și construcții high-level cu placeuri parametrizate. Creat în principal cu scop educațional pentru aprofundarea conceptelor de design al compilatoarelor și modelare formală, instrumentul transformă un limbaj specific într-o reprezentare internă, permițând simularea, generarea de cod și vizualizarea neturilor prin Graphviz. Deși nu include o interfață grafică și nu respectă complet standarde existente precum PNML, proiectul oferă o abordare flexibilă, bazată pe text, potrivită pentru experimente cu modele mari sau complexe. Posibile dezvoltări viitoare includ adăugarea unei interfețe grafice, îmbunătățirea semanticii rețelelor high-level și integrarea cu formate standard.

Abstract

This project presents a command-line tool for parsing and simulating Petri Nets, with support for various extensions such as colored tokens, inhibitor arcs, time constraints, and high-level constructs with parameterized places. Designed primarily for learning about compiler design and formal models, the tool translates a domain-specific language into an internal representation, allowing simulation, code generation, and visualization through Graphviz. While it lacks a graphical interface and full support for existing standards like PNML, the project offers a flexible, text-based approach suitable for experimenting with large or complex models. Potential future work includes adding graphical support, improving high-level net semantics, and integrating standard formats.

Contents

1 Introduction **4**

2 Preliminaries **5**

2.1 Definitions 5

2.2 Current State of Petri Net Modeling 9

2.2.1 Petri Net Markup Language (PNML) 9

2.2.2 Existing Tools and Modeling Platforms 9

2.3 Objectives of the Thesis in Context 10

3 Petri Net Description Language **10**

3.1 Compilation in Haskell 11

3.2 GoLang 12

3.3 Implementation 13

3.3.1 The PNDL Language 13

3.3.2 The Interface of the Project 14

3.3.3 Compile command 14

3.3.4 Generate-Go commnand 15

3.3.5 GraphViz Generation 17

3.4 Comparison with Existing Tools 18

3.5 Testing the Project with Examples 19

3.5.1 Inhibited Arcs 19

3.5.2 Coloured tokens 19

3.5.3 Timed transitions 19

3.5.4 Highlevel nets 19

3.6 Testing the Limits of the Tool 20

3.6.1 Parametrized Expansion Overhead 20

3.6.2 Integer Overflow in Go Backend 20

4 Conclusion **21**

Shortcomings of the Project 21

Overview and Future Work 22

Appendix **26**

1 Introduction

This thesis is both a research and software development project, situated at the intersection of the domains of Petri net theory and compiler design. It explores the use of formal methods for the modeling and simulation of concurrent systems, aiming to bridge the gap between theoretical models and executable implementations.

At the core of this work lies the general concept of Petri nets, a formal mathematical tool used to represent and analyze concurrent, distributed, and parallel systems. Petri nets model system behavior using a combination of places, transitions, and tokens, providing a framework for analyzing crucial system properties such as deadlocks, reachability, and synchronization. Their structured nature makes them especially useful for reasoning about complex system dynamics in a clear and analyzable way.

To support automated analysis and execution, this thesis integrates compiler techniques, highlighting the role of compilers in transforming high-level Petri net descriptions into executable code. By combining the rigor of Petri-net theory with the systematic transformation capabilities of compilers, this work proposes a domain-specific language that allows users to describe Petri nets in a structured and readable format. This language is then compiled into executable representations, facilitating the modeling, simulation, and visualization of concurrent systems.

The motivation behind choosing this topic stems from both a theoretical interest in formal systems and a practical desire to support verification and testing of concurrent and parallel systems. As distributed software becomes increasingly prevalent, tools that simplify the specification and analysis of such systems are of growing importance. A dedicated language for Petri nets supports this goal by lowering the barrier to entry for formal modeling and enhancing the efficiency of simulation and validation processes.

The main contribution of this thesis is the development of a domain-specific language for describing Petri nets, along with a compiler implemented in Haskell. This compiler translates Petri net descriptions into GoLang code for execution and GraphViz for graphical visualization, enabling both simulation and documentation. The project draws on knowledge from formal language theory,

compiler construction, and concurrent system modeling, offering a cohesive and practical toolset for analyzing distributed behaviors.

Why Petri Nets

One of the main motivations for choosing a project on Petri nets was my interest in distributed systems. These systems are inherently complex due to concurrency, synchronization, and communication challenges. Petri nets, with their formal semantics and graphical nature, provide a powerful framework for modeling and analyzing such systems. Their ability to represent parallelism and detect issues like deadlocks or race conditions makes them highly relevant for the study and simulation of distributed architectures.

Additionally, at the time I selected this topic, I was curious about formal verification methods and wanted to explore how mathematical models could be used to reason about the correctness of systems. Petri nets offered an accessible and visually intuitive gateway into the world of formal methods. By working on this project, I was able to gain hands-on experience with the modeling and analysis of concurrent behavior, while also exploring how these models can be transformed into executable simulations, bridging the gap between theoretical models and practical implementations.

2 Preliminaries

2.1 Definitions

The following two definitions are from [1]:

Definition 1. *A **net** is a triple $N = (P, T, F)$ where:*

1. *P and T are finite disjoint sets of **places** and **transitions**.*
2. *$F \subseteq (P \times T) \cup (T \times P)$ is a set of flow relations.*
3. *for every $t \in T$, there exists $p, q \in P$ such that $(p, t), (t, q) \in F$.*

4. *for every $t \in T$ and $p, q \in P$, if $(p,t),(t,q) \in F$, then $p \neq q$.*

For a net $N = (P, T, F)$ and $t \in T$, we denote $\bullet t$ the incoming arcs into t and $t\bullet$ the outcoming arcs from t. The notation is analogous for $p \in P$.

Definition 2. *A **Petri net** is a net of the form $\mathrm{PN} = (N, M, W)$, where:*

1. *$N = (P, T, F)$ is a net.*
2. *$M : P \to Z$ is a place multiset, where Z is a countable set. M maps for every place the number of tokens it has, and it is known as the **initial marking**, noted with M_0 .*
3. *$W : F \to Z$ is an arc multiset, which denotes the arc's weight.*

The following definition from [2] describes the conditions for firing a transition, how a new marking M' is created and what it is its relationship to the previous marking M.

Definition 3. *Having a Petri net $PN = (N, M, W)$ with $N = (P, T, F)$:*

1. *A transition $t \in T$ can be **fired** if for every arc $a = (p,t) \in \bullet t$, $W(a) \leq M(p)$.*
2. *By firing t, a new marking M' is created as follows:*

$$M' = \{(p, x) |\ p \in P,\ x = M(p) - W((p,t)) + W((t,p))\}$$

   If $(p,t) \notin T$ or $(t,p) \notin T$, then $W(p,t)$, respectively $W(t,p)$ will be subtituted with 0, or its equivalent in Z.

3. *If M' can be obtained by firing a transition from M, then this relation is noted $M \to M'$.*
4. *If M' can be obtained by firing an arbitrary number of transitions starting with M, then M' is reachable from M, and reachability is noted like this: $M \xrightarrow{*} M'$.*

The following definition from [3] defines Coloured Petri Nets, which extend the definition of simple PN by allowing tokens to carry data values, called colours, enabling more compact and

expressive models. Instead of using multiple places and transitions to represent different types of information or behavior, CPNs use colour sets and expressions to distinguish between token types and define complex conditions for transition firing.

Definition 4. *A **coloured Petri net (CPN)** is defined by a tuple* $\mathrm{CPN} = (N, C, M, W)$ *where:*

1. $N = (P, T, F)$ *is a net.*
2. C *is the set of colour classes.*
3. $M : P \to 2^{C \times Z}$ *is the modified PN marking function to accomodate the use colours*
4. $W : F \to 2^{C \times Z}$ *is the modified PN arc weight function to accomodate the use of colours*

The following definition from [2] introduces Petri Nets with inhibitor arcs, which extend the definition of simple PN by allowing transitions to be connected to places via inhibitor arcs. These arcs enable a transition to fire only when the connected place contains less tokens than them, allowing the modeling of non-monotonic behavior and enabling control based on the absence of conditions. This extension supports more expressive system descriptions, particularly for capturing mutual exclusion, conditional execution, and resource constraints in concurrent and distributed systems.

Definition 5. *A **Petri net with inhibitor arcs** is defined by a tuple* $\mathrm{PNI} = (\mathrm{PN}, \mathrm{Inh})$ *where:*

1. PN *is a Petri net using the above notation*
2. Inh *is the inhibition matrix defined in* $(Z_\omega \backslash 0)^{P \times T}$.
3. *To the firability criteria of a transition* $t \in T$ *from a marking* M*, the condition of* $M < Inh(t)$*(the comparison is made component per component) is added.*

The following definition from [2] presents Timed Petri Nets, which extend the definition of simple PN by associating time semantics with transitions or tokens. This addition enables the modeling of temporal constraints and delays in system behavior, allowing transitions to fire only after

specific time intervals have elapsed. Timed Petri Nets are particularly useful for representing real-time systems, performance analysis, and scenarios where timing and synchronization are critical to system correctness.

Definition 6. *Timed Petri nets extend Petri nets by associating a firing duration with each transition. More formally, a* ***timed Petri net*** *is a tuple* $\mathrm{TPN} = (\mathrm{PN}, \mathrm{IS})$*, in which* PN *is a Petri net, and* $\mathrm{IS} : T \to Q^+ \times (Q^+ \cup \{\infty\})$ *is the static interval function.*

The following definition from [2] introduces *High-Level Petri Nets*, a term we adopt to describe what are formally known as *Parameterized Petri Nets*. These extend basic Petri nets by enabling parts of the net to be specified as parameterized templates. Such templates use indexed variables like <N> to define scalable sections that are automatically expanded into N structurally identical subnetworks. Places, transitions, and arcs involving these parameters are replicated per instance, allowing concise modeling of systems with repeating components such as multiple clients or concurrent processes. This approach supports modular design and simplifies the specification of large, regular Petri nets.

Definition 7. *A* parameterized Petri net (PPN) *is a tuple*

$$\mathrm{PPN} = (P_0, P_{\mathrm{tmpl}}, T_0, T_{\mathrm{tmpl}}, F, M_0)$$

where:

- P_0 *and* T_0 *are sets of base places and transitions.*
- P_{tmpl} *and* T_{tmpl} *are sets of place and transition templates.*
- $F \subseteq (P_0 \cup P_{\mathrm{tmpl}}) \times (T_0 \cup T_{\mathrm{tmpl}})$ *is the arc set.*
- $M_0 : P_0 \cup P_{\mathrm{tmpl}} \to \mathbb{N}$ *is the initial marking.*

Given a replication factor $N \in \mathbb{N}_{>0}$*, the unfolded net* $\mathrm{PPN}(N)$ *is a standard Petri net where each* $p \in P_{\mathrm{tmpl}}$ *and* $t \in T_{\mathrm{tmpl}}$ *is instantiated as* $p_1, \ldots, p_N$ *and* $t_1, \ldots, t_N$*.*

The flow relation and initial marking are lifted accordingly. For example, $(p, t) \in F$ *with* p, t *both templates expands to* (p_i, t_i) *for all* i*.*

2.2 Current State of Petri Net Modeling

2.2.1 Petri Net Markup Language (PNML)

The *Petri Net Markup Language* (PNML) was introduced as an attempt to standardize the representation of Petri Nets in a structured, tool-independent way. As described by Weber and Kindler [4], PNML defines an XML-based format designed to encode various classes of Petri Nets, including Place/Transition nets, Colored Petri Nets, and more. It aims to facilitate tool interoperability and model exchange between different Petri Net applications and environments.

To date, PNML remains the closest the Petri Net research community has to a widely accepted standard for representing and exchanging Petri Net models. Its structure allows for extensibility and is supported by several modeling tools and academic frameworks. However, due to the complexity and verbosity inherent in XML, human readability and direct manipulation are often limited without dedicated graphical interfaces or tool support.

2.2.2 Existing Tools and Modeling Platforms

Numerous tools exist for modeling, simulating, and analyzing Petri Nets. Notable platforms include:

- **CPN Tools** [5] is a widely-used environment for Colored Petri Nets. It supports graphical modeling and powerful simulation capabilities tailored to high-level Petri Nets.
- **Petrinet Editor** [6] offers a browser-based modeling interface for basic and extended Petri Net types, with export and visualization support.
- A comprehensive and regularly updated list of Petri Net tools is maintained by the University of Hamburg [7]. This database includes editors, analyzers, simulators, and verification tools for a wide spectrum of Petri Net variants.

Despite the variety of existing solutions, many tools target specific net types or modeling scenarios and may lack support for extensible, user-defined features such as parametric expansion, custom net transformations, or integration into modern language tooling (e.g., JSON, Graphviz, or Go code generation). These gaps motivate further exploration into new modeling approaches and DSLs tailored for flexible and composable Petri Net representations.

2.3 Objectives of the Thesis in Context

The main goal of this thesis is to develop a domain-specific language for defining Petri nets, along with a compiler that translates these definitions into executable code for simulation and LaTeX code for visualization. The specific objectives are:

- To design a simple and expressive language for defining Petri nets.
- To implement a compiler in Haskell that generates both simulation-ready code and graphical representations.
- To compare this approach with existing tools and evaluate its advantages in terms of usability, automation, and flexibility.

This project bridges the gap between formal modeling and implementation, aiming to make Petri Net analysis not only more accessible but also automated.

3 Petri Net Description Language

We start a few preliminary notions on the compilation in Haskell, a short presentation motivation the usage of the programming language Go. Then we describe how the DSL of this thesis was defined and how it was implemeted and tested.

3.1 Compilation in Haskell

Definition 8. *A compiler is a program that translates source code written in one language into another form, typically a lower-level language.*

Compilers are usually confused with transpilers or interpreters. The main difference between a compiler and a transpiler is that the former translates source code from one language to another without lowering the abstraction level, whilst the first one usually lowers the abstraction level. An example of the usage of transpilers is related to TypeScript language which is a superscript of JavaScript. The difference between a compiler and an interpreter consists in the time at witch the translation is done. In the case of a compiler, the translation is done prior to execution, while in the case of an interpreter, the code is translated line by line at runtime. For instance, interpreters are used for the Python programming language, and Java has a compiler-interpreter hybrid. [8]

Definition 9. *Syntactic analysis, or parsing, is the compiling phase that processes a sequence of tokens (usually from the lexical analyzer) and produces a structured representation—often an abstract syntax tree (AST)—based on the grammar of the source language.*

Traditionally, syntactic analysis is performed using classical parsers, such as top-down (LL) or bottom-up (LR, LALR) algorithms. These parsers are typically generated by tools like Yacc or Bison, based on a formal grammar written in a declarative notation such as BNF. The grammar is separated from the implementation, and the process often requires an accompanying lexer (e.g., Flex) to provide tokens. [8]

In contrast, parser combinators [9] offer a more modern and expressive approach, especially in functional programming languages like Haskell. A parser combinator library allows the programmer to define parsers directly in code, using higher-order functions to build complex parsers from simpler ones. This eliminates the need for external grammar files and generation steps. In Haskell, the most widely used library for this purpose is Parsec [10].

Parsec allows grammars to be embedded directly into Haskell source code, making the syntax of the language itself a first-class citizen. This style supports powerful abstraction, composability,

and precise error handling, as each parser is a value and can be passed around, combined, or reused. Furthermore, it enables fine-grained control over the parsing process, and benefits from Haskell's strong static type system.

In this project, the syntax of the Petri Net description language is implemented using Parsec. The choice of Parsec over a traditional parser generator was motivated by its flexibility and seamless integration with Haskell's data structures. The custom language that we define supports advanced constructs, such as parametrized high-level net sections, which can be naturally represented and parsed using Parsec's compositional design.

3.2 GoLang

Go (or Golang) [11] is a statically typed, compiled programming language developed at Google, designed for simplicity, performance, and concurrency, as discussed by Pike [12]. Its usage in this project is motivated by several key features that align well with the requirements of simulating and executing Petri nets.

Firstly, Go offers native support for concurrency through *goroutines* and *channels* [12]. Since Petri nets are inherently concurrent models, simulating them efficiently requires lightweight thread-like constructs and synchronization mechanisms. Go's model of concurrent execution, inspired by Hoare's Communicating Sequential Processes (CSP) [13], maps naturally to the semantics of Petri nets, making the implementation of concurrent behavior both elegant and efficient. Even though I did not use this feature of the language in the project, it can be extended in the future to include features that would benefit from it, like cicle detection, or the firing of multiple independent transitions at once.

Secondly, Go has a minimalistic and clean syntax, which improves readability and maintainability. For a simulation backend intended to be easily extensible and understandable, avoiding the complexity of languages like C++ or Java is a significant benefit. Furthermore, the absence of a complicated type system or inheritance hierarchy in Go simplifies the representation of tokens, transitions, and execution logic.

Thirdly, Go produces statically linked, self-contained binaries, facilitating easy deployment of the simulation tool without runtime dependencies [12]. This makes it ideal for building tools that are meant to be used in varied environments, especially in academic or distributed computing setups where installation overhead should be minimal.

Lastly, Go has a growing ecosystem and active community, with tools for performance profiling, testing, and package management. It is increasingly used in systems programming, making it a pragmatic choice that balances low-level control with developer productivity.

In conclusion, Go's design principles, concurrency model, ease of deployment, and simplicity make it particularly well-suited for the execution layer of a Petri net simulation system. These factors justify its selection over alternatives such as Python (limited concurrency performance), Java (heavier runtime), or C++ (steeper learning curve and more complex concurrency primitives).

3.3 Implementation

3.3.1 The PNDL Language

To support the modeling of various types of Petri Nets in a flexible and extensible way, I designed a custom domain-specific language called *Petri Net Description Language* (PNDL). This language is used to define Petri Nets in a human-readable and modular format, supporting extensions such as colored tokens, timing constraints, inhibitor arcs, and high-level parameterized components. The syntax of PNDL is intentionally minimal and declarative, using sections like `NET_TYPE`, `VARIABLES`, `PLACES`, `TRANSITIONS`, and `ARCS` to describe the structure and behavior of the net. Figure 1 in the Appendix presents the core syntax of PNDL, along with inline comments that clarify the semantics and conditional requirements for each section based on the net type. This format serves as the input to the compiler I developed, which translates PNDL specifications into JSON and Graphviz representations.

3.3.2 The Interface of the Project

The final implementation of the parser program has a CLI. The interface has three commands: `compile`, `generate-go` and `graphviz`. Every command takes as argument a file written in the format described in the next subsections. In the further sections of this chapter, the behavior and implementation details will be described.

3.3.3 Compile command

To interpret a PNDL file, the `compile` command invokes a series of functions that parse its contents using the `Parsec` module. These functions structure the parsed data into custom Haskell objects, which are then translated into JSON, depending on the type of Petri Net being described. The compilation process supports several Petri Net extensions, as follows:

1. **Simple Petri Nets:**
    - The initial marking is stored in a dedicated object that maps each place name to its corresponding token count in M_0.
    - Each place is represented by an object containing a placeholder attribute, allowing future extensions such as dynamic modification of the place structure.
    - Arcs are grouped by transition and direction (input or output). Each arc includes a `weight` attribute.
2. **Coloured Petri Nets:**
    - Token counts and arc weights are represented as maps from color names represented strings to integer values, instead of plain integers.
3. **Timed Petri Nets:**
    - Each transition object includes two additional integer attributes: `minTime` and `maxTime`, defining the transition's firing interval.

4. **Inhibited Petri Nets:**

   - Transitions may optionally include an `inh` attribute for inhibitor arcs. This attribute is an integer for simple nets, or a color-to-count map for coloured nets.

5. **High-Level Petri Nets:**

   - Places defined with a parameter (e.g., `Place<N>`) are expanded into multiple concrete places: `Place_1`, `Place_2`, ..., `Place_N`, each initialized with the same token values.
   - Each generated place `Place_X` includes an entry in the initial marking such as {`Variable_X: 1`}. For non-coloured nets, all tokens are tagged with a default color: `token`.
   - Transitions that are not connected to high-level places are replicated for every combination of parameter values. For every variable V, this results in transitions like:

$$T\_V\_1, \quad T\_V\_2, \quad \ldots, \quad T\_V\_n$$

   - Transitions that connect to high-level places are only expanded based on the variables associated with those specific places.
   - Each generated transition `T_Vx_y` has an entry {`Vx_y: 1`} added to every weight map in its arcs, indicating the specific instance it operates on.
   - All transformations related to high-level Petri Nets are performed entirely at compile-time. The resulting JSON, and subsequently the Go code, treats the net as if it were a simple, non-parametrized Petri Net.

3.3.4 Generate-Go commnand

The `generate-go` command interprets only the `NET_TYPE` directive from the PNDL file and produces a fully functional Go source file based on a predefined template. This template is almost identical to a standard Go file, with the exception of custom compile-time conditionals expressed

using the syntax `<<condition, code>>`. These conditional code blocks are evaluated during generation and are only included in the final output if their corresponding condition holds. Conditions can be logical combinations such as `COLORED`, `NOT COLORED`, `TIMED`, `INHIBITED`, or even composite conditions like `TIMED AND INHIBITED`.

The structure of the generated Go file is modular and follows the logic of simulating a Petri Net execution:

- The **data types** (`Arc`, `Transition`, `Marking`, `Net`, and `JSONInput`) are defined first. These types adapt to the net type using the conditional syntax. For example, in a COLORED net, arc weights and initial markings are maps from color names to integers, while in a simple net, they are plain integers.

- The **JSON loader** reads the net structure and initial marking from `net.json`, processes the color set (if applicable), and fills in any missing places with zero tokens. For TIMED nets, default firing intervals are ensured to be at least 1 if not explicitly set.

- The **arc comparison function** (`compareArc`) evaluates whether a transition is enabled by comparing the current marking with the arc's weight and inhibitor values. Different versions of this logic are compiled depending on the combination of COLORED and INHIBITED settings.

- The **main simulation loop** repeatedly computes the set of viable transitions, selects one randomly, and fires it. For TIMED nets, the logic becomes more complex: transitions are started and appended to an `activeTransitions` list, which tracks remaining execution time. When a transition's timer expires, its output arcs are applied.

- The simulation continues until no more transitions are viable and no timed transitions are pending (in TIMED mode).

- Additional helper functions handle tasks such as transition selection, applying input/output arcs, and managing time progression.

All code variants are embedded in the template using the `<<condition, code>>` syntax. During generation, the system performs a textual transformation, resolving all such blocks based on the current Petri Net type. This ensures that the final Go program is minimal, readable, and free of unused logic paths.

3.3.5 GraphViz Generation

GraphViz is a graph visualization tool written in dot syntax, that can be easily compiled in a LaTeX environment. This representation can be exported in all sorts of formats like PDF, PNG, JPEG and many more.

The GraphViz module of the project is responsible for producing a .dot representation of a Petri Net. Its modus operandi closely mirrors that of the `toJSON` Parsec-based parser. It also relies on a Parsec parser to interpret the PNDL input and generate the corresponding arguments for the `DOT` syntax. The structure of the parser and its logic for handling `COLORED`, `TIMED`, and `INHIBITED` nets are analogous to those used in the JSON generator.

The key distinction arises when dealing with `HIGH-LEVEL` nets. In this case, high-level constructs such as variable-bound places and transitions are not expanded into all possible concrete instances. Instead, they are grouped according to the variable on which they depend. Each such group is rendered as a distinct subgraph (or `subgraph cluster`) within the overall DOT graph. This visual separation serves to highlight the parametric nature of the net and the dependencies of each group on its associated variable.

All remaining non-parametric (simple) places and transitions are placed in a special group labeled `simple`, representing components independent of any variable. This structured organization enhances readability and provides insight into the modular, parameterized design of high-level nets.

In the generated GraphViz graph, places and transitions are visually distinguished using different node formatting styles. Places are represented as ellipses, while transitions are drawn as rectangles. This differentiation aligns with common Petri Net visualization conventions and improves the interpretability of the diagram.

3.4 Comparison with Existing Tools

The domain of Petri Net modeling has seen the development of various tools and standardization efforts. Among these, the *Petri Net Markup Language (PNML)* [4] represents the most widely accepted attempt at a standardized representation format. PNML is designed for interoperability and exchange of models between tools, but its XML-based structure can become verbose and less suitable for fast prototyping.

Graphical modeling environments such as *CPN IDE* [5] and *PetriNet Editor* [6] offer visual construction, simulation, and in some cases, code generation capabilities. These tools focus on ease of use for system designers and support a wide range of Petri Net extensions. Additionally, the *Petri Nets Tools Database* [7] catalogs a wide variety of tools supporting analysis, modeling, and transformation of Petri Nets.

In contrast, the project presented in this thesis introduces a textual, compiler-inspired pipeline for modeling Petri Nets. One of its main contributions is a high-level modeling mechanism that supports *parameterized nets*, allowing users to define templates with indexed variables. This supports concise descriptions of structurally repetitive systems, improving scalability and modularity. Unlike CPN IDE or PetriNet Editor, which support simulation and GUI modeling, this tool is optimized for textual specification, batch processing, and automatic code generation.

Furthermore, the tool integrates support for extensions such as *Colored*, *Timed*, and *Inhibited* Petri Nets within a unified syntax. This composability is particularly useful in automated or programmatic settings. However, it currently lacks several features provided by mature tools:

- Only textual simulation is supported via the Go backend; lacks visual or interactive analysis tools.
- There is no graphical modeling interface, which may hinder adoption by users unfamiliar with textual DSLs.
- It does not support PNML import/export, limiting interoperability with standardized ecosystems.

Despite these limitations, the tool excels in structural scalability and automation. It is particularly well-suited for use cases involving programmatically generated models, code integration, or parameter sweeps, and thus serves as a complementary approach to existing GUI-based and simulation-focused tools.

3.5 Testing the Project with Examples

3.5.1 Inhibited Arcs

Listing 3 in the Appendix describes a net with an inhibited arc. Its graph is shown in Figure 4, demonstrating that the GraphViz output can support this type of arcs. In Figure 5, the steps of the execution of the net can be seen. We can observe how the program stops when the `Consumer` place has 5 tokens.

3.5.2 Coloured tokens

Listing 4 describes the same net as before but has coloured tokens. It consumes blue and yellow tokens and produces green tokens, as seen in Figure 6. The output stops when the consumer has 5 green tokens, as seen in Figure 7.

3.5.3 Timed transitions

Listing 5 describes the same net as before but has timed transitions. This feature does not impact the GraphViz output. When running the generated GoLang code, an output similar to Listing 6 is generated. The output can only be similar because Timed Nets introduce a randomised element, the duration of the transitions.

3.5.4 Highlevel nets

Listing 1 describes a high level net, with the difference that here N is instanciated as 100. Because a highlevel net does not change the behavior of the GoLang code, its output isn't included. In

contrast, the GraphViz representation of this net has a major difference from the above discussed representations, and can be seen in Figure 8. That difference is the presence of the gray rectangle surrounding the places and transitions that depend on the variable N.

3.6 Testing the Limits of the Tool

To explore the performance boundaries and correctness of the Petri Net toolchain, a series of stress tests were designed and executed. These tests targeted both the parser and the backend code generation components.

3.6.1 Parametrized Expansion Overhead

One approach for evaluating scalability involves defining a **High-Level Petri Net** with an extremely large parameter. The network described in Listing 1 models a client-server interaction, parameterized over $N = 1{,}000{,}000$ clients, and incorporates `TIMED`, `INHIBITED`, `COLORED`, and `HIGH_LEVEL` features.

Execution of this definition resulted in the operating system terminating the process due to excessive memory consumption or CPU overload. This indicates that the tool currently encounters performance bottlenecks when expanding high-cardinality parametrized definitions. The corresponding error is shown in Figure 2. The exact value of N that triggers this error varies depending on the machine's hardware specifications. In the case of my machine which had around 11.5 free GiB of RAM, the problems started showing up for $N > 700{,}000$

3.6.2 Integer Overflow in Go Backend

A separate test focused on identifying numeric limitations in the backend code generation. The network described in Listing 2 utilizes large token weights to provoke overflow behavior in the Go language implementation.

This net successfully compiles and passes through the parser. However, execution of the generated Go code results in overflow, as the default integer type on the target system is a 64-bit signed

integer, with a maximum representable value of $2^{63} - 1$. This exposes a limitation in the type safety of the backend generator and suggests a need for improved handling of numeric ranges.

Figure 3 shows the behaviors of the Go script for the first 10 iterations (plus the starting marking) of the net. We can observe how the places sometimes have negative values, a clear sign of integer overflow.

4 Conclusion

Shortcomings of the Project

This project, while functional in its core objectives, has several limitations when compared to existing tools in the domain.

Missing Features Present in Other Tools

A number of features commonly found in similar tools are not implemented in this project:

- **Graphical Interface**: The tool lacks a graphical editor or visualization interface, unlike tools such as CPN Tools [5] or PetriNet Editor [6]. This omission stems from the project's primary goal: learning about compiler construction and language design. As such, efforts were focused on parsing, representation, and code generation rather than UI development.
- **Standard Integration**: The tool does not natively support the Petri Net Markup Language (PNML) [4], which is widely regarded as a standard. This was largely due to late discovery of the format during the development process. A future version of the project could integrate support for PNML import/export to improve interoperability.

Limitations in the High-Level Net Implementation

The high-level Petri Net implementation introduces several structural restrictions:

- **No Cross-Instance Arcs**: The tool does not support transitions connecting instances of different indices in high-level places. For example, a transition with an incoming arc from `P<i>` and an outgoing arc to `P<j>` (where $i \neq j$) is disallowed, due to the logic used in the unfolding process.

- **Uniform Initial Marking**: All instances of a high-level place `P<N>` must have the same initial marking. It is not possible to specify distinct initial token distributions for different values of i within the same parameterized place.

- When more than one group of high-level places interacts within the same net, the resulting behaviour becomes undefined. The current expansion logic does not resolve interactions across distinct parameter groups, leading to uncertain semantics.

Overview and Future Work

The development of this project was driven by a desire to explore compiler design principles through the lens of a domain-specific language for modeling Petri Nets. The goal was to support advanced features such as high-level constructs, colored tokens, timed behavior, and inhibitor arcs, and to compile these into executable code and visualization formats. The implementation focused on correctness and extensibility, with an emphasis on the backend logic rather than user interface aspects.

Future improvements could include the development of a graphical user interface (GUI), support for simulation interactivity, formal verification tools, and alignment with existing standards such as the Petri Net Markup Language (PNML). Additionally, addressing current limitations of high-level net support, such as interactions between multiple parameterized groups, could significantly enhance the tool's expressiveness and reliability.

From a personal standpoint, this project provided valuable insight into language parsing, transformation pipelines, and system modeling. The results achieved are significant in that they enable efficient modeling of complex systems using a concise and modular syntax. While not aiming

to compete directly with mature tools in the field, this project represents a meaningful proof of concept and an educational endeavor with tangible outcomes.

In terms of practical applications, the tool can be used in academic environments for teaching formal methods or as a lightweight modeling solution for prototyping concurrent systems. With continued development, it has the potential to serve as a flexible framework for Petri Net analysis in specialized domains such as distributed systems, protocol verification, or embedded systems design.

References

[1] G. Rozenberg and J. Engelfriet, "Elementary net systems," in *Advanced Course on Petri Nets*. Springer, 1996, pp. 12–121.

[2] M. Diaz, *Petri nets: fundamental models, verification and applications*. John Wiley & Sons, 2013.

[3] C. Girault and R. Valk, *Petri nets for systems engineering: a guide to modeling, verification, and applications*. Springer Science & Business Media, 2013.

[4] M. Weber and E. Kindler, "The petri net markup language," in *Petri net technology for communication-based systems: advances in Petri nets*. Springer, 2003, pp. 124–144.

[5] E. Verbeek and D. Fahland, "Cpn ide: An extensible replacement for cpn tools that uses access/cpn," in *3rd International Conference on Process Mining, ICPM 2021*. CEUR-WS. org, 2021, pp. 29–30.

[6] E. Kučera, O. Haffner, P. Drahoš, R. Leskovský, and J. Cigánek, "Petrinet editor + petrinet engine: New software tool for modelling and control of discrete event systems using petri nets and code generation," *Applied Sciences*, vol. 10, no. 21, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/21/7662

[7] Theoretical Foundations Group (TGI), "Petri nets tools database," https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/complete_db.html, 2025, [accessed: 2025-06-05].

[8] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers principles, techniques & tools*. pearson Education, 2007.

[9] G. Hutton and E. Meijer, "Monadic parser combinators," 1996.

[10] https://hackage.haskell.org/package/parsec, [accessed: 2025-06-12].

[11] https://go.dev/, [accessed: 2025-06-12].

[12] R. Pike, “Go at google,” in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, 2012, pp. 5–6.

[13] S. D. Brookes, C. A. Hoare, and A. W. Roscoe, “A theory of communicating sequential processes,” *Journal of the ACM (JACM)*, vol. 31, no. 3, pp. 560–599, 1984.

Appendix

```
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:0
   yellow:0] Producer:map[blue:2 green:0 yellow:2]]}
TIME:  1
Started transition Produce with id: 0 with duration 2
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:0
   yellow:0] Producer:map[blue:1 green:0 yellow:1]]}
TIME:  2
Started transition Produce with id: 1 with duration 3
Ended transtition Produce with id: 0
Ended transtition  with id: 0
{map[Buffer:map[blue:0 green:1 yellow:0] Consumer:map[blue:0 green:0
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  3
Started transition Consume with id: 2 with duration 3
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:0
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  4
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:0
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  5
Ended transtition Consume with id: 2
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:1
   yellow:0] Producer:map[blue:1 green:0 yellow:1]]}
TIME:  6
Started transition Produce with id: 3 with duration 2
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:1
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
```

Figure 1: PNDL syntax

```
~NET_TYPE
COLORED | TIMED | INHIBITED | HIGH_LEVEL
// Specify one or more net extensions
// If the net is simple, this section can be ommited

~VARIABLES
VariableName = Value
// Optional variables used for parametrization
// If the net is not high level, this section can be ommited

~PLACES
PlaceName[<Variable>] [tokens={color1:count, color2:count, ...}]
// For coloured nets
PlaceName[<Variable>] [tokens=count]
// For simple nets
// Tokens field is optional
// In its absence, the m0 will be filled with 0

~TRANSITIONS
TransitionName [time=(min,max)]
// Optional time interval for timed nets
// If absent, then time will default to (1,1), for timed nets

~ARCS
Source -> Target [weight=value] [inh=value]
// Only one of source or target can be a place
// Only one of source or target can be a transition
// If source is a transition, then the arc is an output of Source
// If Target is a transition, then the arc is an input of Target
// The weight field is always present
// Value can be replaced with an integer or a color-value map
// The inh field is ommited if the net doesn't have inhibitor arcs
// If the net has inhibitor arcs this field is optional
// If Source or Target is a high level place then
//the <Variable> of the place needs to be present
```

```
~NET_TYPE
TIMED INHIBITED HIGH_LEVEL COLORED
~VARIABLES
N = 1000000   // Number of clients
~PLACES
ClientIdle<N> [tokens={blue:1}]
RequestSent<N> [tokens={}]
ServerIdle [tokens={green:2}]
Processing<N> [tokens={}]
ResponseReady<N> [tokens={}]
ClientReceived<N> [tokens={}]

~TRANSITIONS
SendRequest [time=(1,2)]
ProcessRequest [time=(2,4)]
SendResponse [time=(1,2)]
ReceiveResponse [time=(1,2)]

~ARCS
ClientIdle<N> -> SendRequest [weight={blue:1}] [inh={blue:1}]
SendRequest -> RequestSent<N> [weight={blue:1}]
RequestSent<N> -> ProcessRequest [weight={blue:1}]
ServerIdle -> ProcessRequest [weight={green:1}]
ProcessRequest -> Processing<N> [weight={blue:1}]
ProcessRequest -> ServerIdle [weight={green:1}]
Processing<N> -> SendResponse [weight={blue:1}]
SendResponse -> ResponseReady<N> [weight={blue:1}]
ResponseReady<N> -> ReceiveResponse [weight={blue:1}]
ReceiveResponse -> ClientReceived<N> [weight={blue:1}]
ReceiveResponse -> ClientIdle<N> [weight={blue:1}]
```

Listing 1: Code generating memory errors

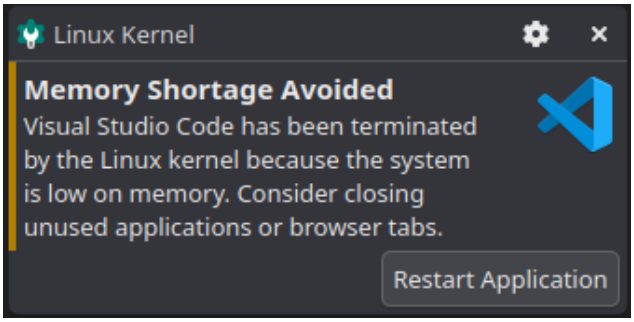

Figure 2: The error message

```
~PLACES
P1 [tokens=1000000]
P2 [tokens=0]
~TRANSITIONS
T1
T2
~ARCS
P1 -> T1 [weight=1]
T1 -> P2 [weight=9223372036854775807]
P2 -> T2 [weight=1]
T2 -> P1 [weight=9223372036854775807]
```

Listing 2: Code generating memory errors

```
➜  PN git:(main) ✗ go run PN.go
{map[P1:1000000 P2:0]}
{map[P1:999999 P2:9223372036854775807]}
{map[P1:999998 P2:-2]}
{map[P1:999997 P2:9223372036854775805]}
{map[P1:-9223372036853775812 P2:9223372036854775705]}
{map[P1:999995 P2:9223372036854775605]}
{map[P1:-9223372036853775814 P2:9223372036854775505]}
{map[P1:999993 P2:9223372036854775405]}
{map[P1:-9223372036853775816 P2:9223372036854775305]}
{map[P1:999991 P2:9223372036854775205]}
{map[P1:-9223372036853775818 P2:9223372036854775105]}
{map[P1:999989 P2:9223372036854775005]}
```

Figure 3: The generated markings by the net in Listing 2

```
~NET_TYPE
INHIBITED
~PLACES
Producer [tokens=1]
Consumer [tokens=1]
Buffer [tokens=0]

~TRANSITIONS
Produce
Consume

~ARCS
Producer -> Produce [weight=1]
Produce -> Buffer [weight=1]
Buffer -> Consume [weight=1]
Consume -> Consumer [weight=1] [inh=5]
Consume -> Producer [weight=1]
```

Listing 3: Code testing inhibitor arcs

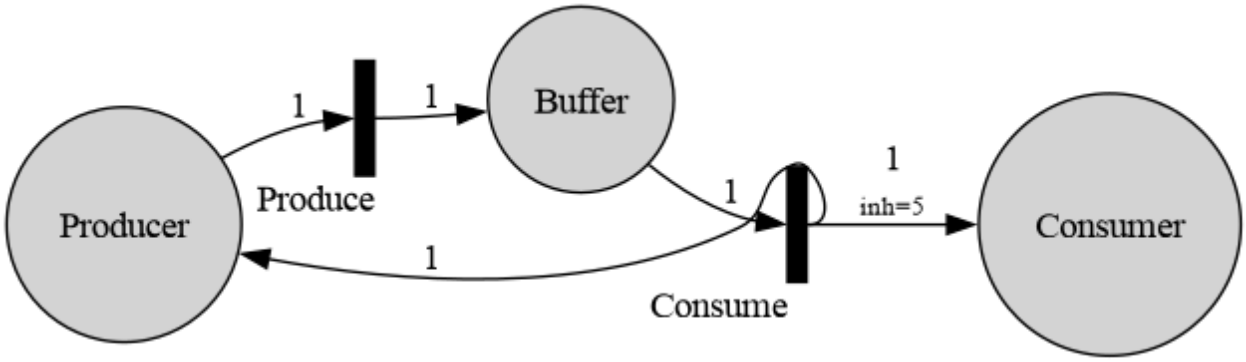

Figure 4: The generated net in Listing 3

```
➜  Parser git:(main) ✗ go run output.go
{map[Buffer:0 Consumer:1 Producer:1]}
{map[Buffer:1 Consumer:1 Producer:0]}
{map[Buffer:0 Consumer:2 Producer:1]}
{map[Buffer:1 Consumer:2 Producer:0]}
{map[Buffer:0 Consumer:3 Producer:1]}
{map[Buffer:1 Consumer:3 Producer:0]}
{map[Buffer:0 Consumer:4 Producer:1]}
{map[Buffer:1 Consumer:4 Producer:0]}
{map[Buffer:0 Consumer:5 Producer:1]}
{map[Buffer:1 Consumer:5 Producer:0]}
There are no more viable transitions
```

Figure 5: The generated output for the net in Listing 3

```
~NET_TYPE
INHIBITED COLORED
~PLACES
Producer [tokens={yellow:1,blue:1}]
Consumer [tokens={green:0}]
Buffer [tokens={}]

~TRANSITIONS
Produce
Consume

~ARCS
Producer -> Produce [weight={yellow:1,blue:1}]
Produce -> Buffer [weight={green:1}]
Buffer -> Consume [weight={green:1}]
Consume -> Consumer [weight={green:1}] [inh={green:5}]
Consume -> Producer [weight={yellow:1,blue:1}]
```

Listing 4: Code testing inhibitor arcs with coloured tokens.

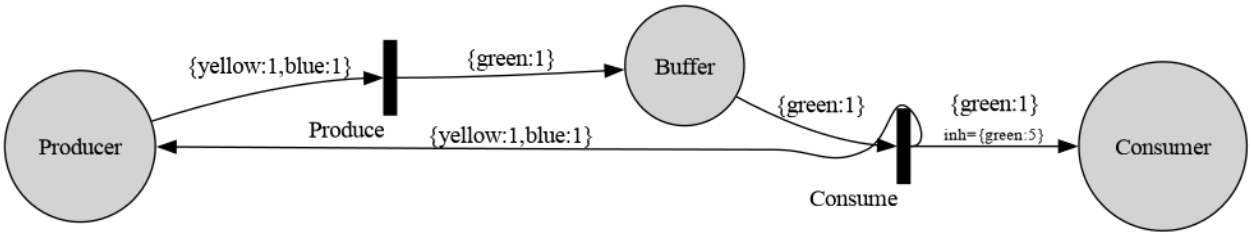

Figure 6: The generated net in Listing 4

```
➜  Parser git:(main) ✗ go run output.go
{map[Buffer:0 Consumer:1 Producer:1]}
{map[Buffer:1 Consumer:1 Producer:0]}
{map[Buffer:0 Consumer:2 Producer:1]}
{map[Buffer:1 Consumer:2 Producer:0]}
{map[Buffer:0 Consumer:3 Producer:1]}
{map[Buffer:1 Consumer:3 Producer:0]}
{map[Buffer:0 Consumer:4 Producer:1]}
{map[Buffer:1 Consumer:4 Producer:0]}
{map[Buffer:0 Consumer:5 Producer:1]}
{map[Buffer:1 Consumer:5 Producer:0]}
There are no more viable transitions
```

Figure 7: The generated output for the net in Listing 4

```
TIME:  7
Ended transtition Produce with id: 3
{map[Buffer:map[blue:0 green:1 yellow:0] Consumer:map[blue:0 green:1
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  8
Started transition Consume with id: 4 with duration 3
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:1
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  9
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:1
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  10
Ended transtition Consume with id: 4
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:2
   yellow:0] Producer:map[blue:1 green:0 yellow:1]]}
TIME:  11
Started transition Produce with id: 5 with duration 2
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:2
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  12
Ended transtition Produce with id: 5
{map[Buffer:map[blue:0 green:1 yellow:0] Consumer:map[blue:0 green:2
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  13
Started transition Consume with id: 6 with duration 3
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:2
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  14
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:2
```

```
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  15
Ended transtition Consume with id: 6
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:3
   yellow:0] Producer:map[blue:1 green:0 yellow:1]]}
TIME:  16
Started transition Produce with id: 7 with duration 3
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:3
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  17
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:3
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  18
Ended transtition Produce with id: 7
{map[Buffer:map[blue:0 green:1 yellow:0] Consumer:map[blue:0 green:3
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  19
Started transition Consume with id: 8 with duration 3
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:3
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  20
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:3
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  21
Ended transtition Consume with id: 8
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:4
   yellow:0] Producer:map[blue:1 green:0 yellow:1]]}
TIME:  22
Started transition Produce with id: 9 with duration 2
```

```
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:4
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  23
Ended transtition Produce with id: 9
{map[Buffer:map[blue:0 green:1 yellow:0] Consumer:map[blue:0 green:4
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  24
Started transition Consume with id: 10 with duration 3
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:4
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  25
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:4
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  26
Ended transtition Consume with id: 10
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:5
   yellow:0] Producer:map[blue:1 green:0 yellow:1]]}
TIME:  27
Started transition Produce with id: 11 with duration 3
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:5
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  28
{map[Buffer:map[blue:0 green:0 yellow:0] Consumer:map[blue:0 green:5
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
TIME:  29
Ended transtition Produce with id: 11
{map[Buffer:map[blue:0 green:1 yellow:0] Consumer:map[blue:0 green:5
   yellow:0] Producer:map[blue:0 green:0 yellow:0]]}
There are no more viable transitions
```

Listing 6: Output resulting from running the net from Listing 5

```
~NET_TYPE
INHIBITED COLORED TIMED
~PLACES
Producer [tokens={yellow:2,blue:2}]
Consumer [tokens={green:0}]
Buffer [tokens={}]

~TRANSITIONS
Produce [time=(2,4)]
Consume [time=(2,4)]

~ARCS
Producer -> Produce [weight={yellow:1,blue:1}]
Produce -> Buffer [weight={green:1}]
Buffer -> Consume [weight={green:1}]
Consume -> Consumer [weight={green:1}] [inh={green:5}]
Consume -> Producer [weight={yellow:1,blue:1}]
```

Listing 5: Code testing inhibitor arcs with coloured tokens and timed transitions.

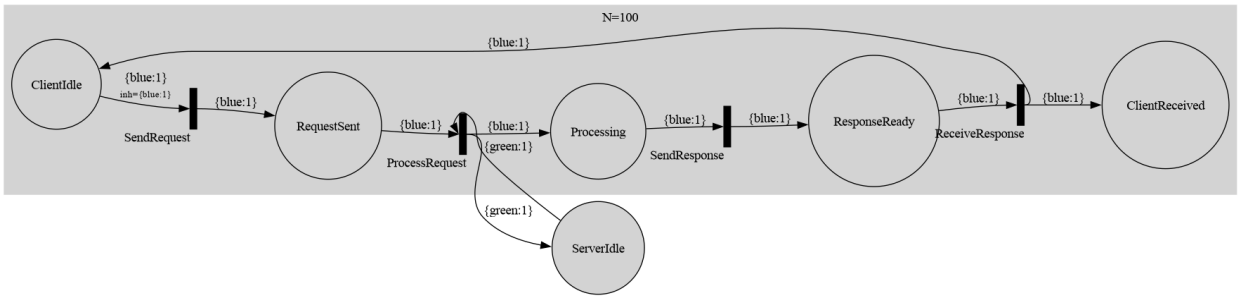

Figure 8: The generated net in Listing 1