1 Introduction

This thesis is both a research and software development project, situated at the intersection 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 topic 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.

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 $PN = (N, M, W)$, 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**, often noted with M_0 or m_0.*
3. *$W : F \to Z$ is an arc multiset, which denotes the arc's weight.*

The following definition 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, and it is from [2]:

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

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 like this:*

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

   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 $M \rightarrow M'$.*

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 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. The definition is taken from [3]:

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

1. *$N = (P, T, F)$ is a net.*

2. *C is the set of colour classes.*

3. *$M : P \rightarrow 2^{C \times Z}$ is the modified PN marking function to accomodate the use colours*

4. *$W : F \rightarrow 2^{C \times Z}$ is the modified PN arc weight function to accomodate the use of colours*

The following definition 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. The definition is taken from [2]:

Definition 5. *A **Petri net with inhibitor arcs** is defined by a tuple $PNI = (PN, Inh)$ 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 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. The definition is taken from [2]:

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

The following definition 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. The definition is taken from [2]:

Definition 7. *A* parameterized Petri net *is a tuple*

$$\mathcal{N} = (P_0, P_{\text{tmpl}}, T_0, T_{\text{tmpl}}, F, M_0)$$

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.*
- $F \subseteq (P_0 \cup P_{\text{tmpl}}) \times (T_0 \cup T_{\text{tmpl}})$ *is the arc set.*
- $M_0 : P_0 \cup P_{\text{tmpl}} \to \mathbb{N}$ *is the initial marking.*

Given a replication factor $N \in \mathbb{N}_{>0}$*, the unfolded net* $\mathcal{N}(N)$ *is a standard Petri net where each* $p \in P_{\text{tmpl}}$ *and* $t \in T_{\text{tmpl}}$ *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 the Specific Subfield

********TBAaaaa

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 aims to bridge the gap between formal modeling techniques and practical implementation, making Petri net analysis more accessible and automated.

3 Contribution

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 the TypeScript language which is a superscript of JavaScript. The difference between a compiler and an interpreter is the time the translation is done. In the case of a compiler the translation is done prior to execution, but in the case of an interpreter, the code is translated line by line at runtime. Interpreters are used for the Python programming language, and Java has a compiler-interpreter hybrid. [4]

Definition 9. *Syntactic analysis, or parsing, is the compiler 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. [4]

In contrast, parser combinators 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.

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 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) is a statically typed, compiled programming language developed at Google, designed for simplicity, performance, and concurrency, as discussed by Pike [5]. Its usage in this project is motivated by several key features that align well with the requirements of simulating and executing Petri nets.

First, Go offers native support for concurrency through goroutines and channels [5]. 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), 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.

Second, 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.

Third, Go produces statically linked, self-contained binaries, facilitating easy deployment of the simulation tool without runtime dependencies [5]. 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 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.

3.4 Implementation

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.

3.4.1 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 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.

3.4.2 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 (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.

3.4.3 Generate-Go commnand

The `generatego` 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.

3.4.4 GraphViz

3.5 GraphViz Generation

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.

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

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

4 Appendix

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
```