A Modular Integrated Development Environment for Coloured Petri Net Models

Sondre Lindaas Gjesdal

Master thesis in Software Engineering

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

June 2023

Western Norway University of Applied Sciences

Abstract

Distributed software systems are becoming increasingly popular and used. Most of modern distributed systems provide the application of concurrency, also including resource sharing, communication and synchronization between different modules. These distributed systems comes with the challenges concerning data synchronization, scalability and performance, among others. By modelling these systems helps with solving these challenges, and there exists multiple tools for this. CPN Tools is one of these tools. CPN Tools is used for editing, simulating and analyzing Coloured Petri nets models. A need has been identified to developed new software for develop new and up to date tools for editing, simulating and analyzing Coloured Petri nets to further development and fit the increasing need for distributed systems. Answering this need, a new simulating tool has been proposed. This thesis proposes an editor focusing on the modelling and visualization with the potentially integrate this simulator. This editor consists of an application running on Electron and using GoJS for modelling. This has resulted in a modelling tool for creating CPN models, with the possibility of increased abstraction of the models of the modern distributed systems.

Acknowledgements

I would like to thank my supervisor Lars Michael Kristensen. It has been an honour learning from you and I appreciate all the advice given throughout this year. You have the ability to clearly deliver information. I have always felt welcome at your office, and greatly appreciated our discussions.

I would also like to thank everyone around me for the help and motivation given.

Contents

Chapter 1

Introduction

1.1 Context and Approach

Distributed systems has been quite popular for some time, and is getting more and more used in the software world. Having a system that can control and or model these are quite beneficial. One of the primary reasons is the increasing demand for scalable, fault-tolerant, and highly available software applications.

Distributed systems allow for the efficient utilization of resources by spreading the workload across multiple interconnected nodes, enabling better performance and handling larger volumes of data. When developing and managing distributed systems, there also comes various challenges and complexities. Some of these challenges include; communication and coordination between different components to achieve a common goal. Ensuring consistent reliable and efficient communication be can become challenging. This also leads to complications with consistency of data synchronization, fault tolerance and resilience, scalability and performance and lastly security and privacy.

Given these challenges, there is a need for tools that can effectively model and control distributed systems. Modeling tools enable system designers and developers to visualize the interactions and behaviors of components within the system, facilitating better understanding and analysis. These tools help identify potential issues, optimize system performance, and validate correctness of the distributed protocols implemented.

CPN Tools [17] is one such tool that can model distributed systems. It has been used to model concurrent systems which is becoming more important now than ever, due to larger, more complicated systems running multiple components needing communication with each other as explained. CPN Tools delivers visualization and analysis to its models for easier understanding.

SFI Smart Ocean[23] is a project that runs from 2020 until 2028 with the goal of developing wireless communications systems to monitor the sea and its ecosystem, and if needed, notify of industrial incidents or changes in living conditions for ocean wildlife.

The goal of this master thesis is to develop a graphical editor for a CPN simulator as a part of the SFI Smart Ocean project. The graphical editor will be a modernization of CPN Tools from 2000. The CPN simulator was made in the spring semester of 2022, through "A Compiler and Runtime Environment for Execution of Coloured Petri Net Models"[12]. The goal of this thesis is to develop a graphical editor to cooperate with the CPN simulator.

Petri nets are mathematical modeling languages widely used for describing and analyzing the behavior of distributed systems. Originally introduced by Carl Adam Petri, who began developing the concept at the age of 13 to explain chemical processes[20], Petri nets have since found widespread application in various domains. At their core, basic Petri nets are directed bipartite graphs consisting of two fundamental elements: places adn transitions. Represented respectively by circles and rectangles. The interaction between places and transitions is visualized through directed arcs connecting them. Places can contain tokens, which are denoted as dots within the respective places.

To extend the capabilities of basic Petri nets, the concept of Coloured Petri Net (CPN)[16], was introduced. CPN allows for the distinction between tokens and provides the ability to attach data to tokens. This extension enhances the expressive power of Petri nets by incorporating data-driven aspects into the modeling process. By associating data with tokens, CPN enables the representation of more complex system behaviors and enables the analysis of scenarios where data values play a significant role.

This report presents a selection of newer frameworks for editing graphical models, with the ultimate goal of utilizing them to edit and run simulations for Coloured Petri Net (CPN). The aim is to explore and evaluate these frameworks to determine their suitability for supporting CPN editing and simulation tasks. We will emphasise modernization and what modern distributed systems require from a CPN editor. By the end of the project, it is expected that these frameworks will provide effective tools for visual modeling and simulation of CPNs.

1.2 SFI Smart Ocean

The SFI Smart Ocean project is focused on developing wireless communication systems to monitor the sea and its ecosystem. The project aims to address various challenges, including the detection and notification of industrial errors and changes in living conditions for ocean wildlife. Industrial errors such as oil spills, chemical leaks, and untreated sewage discharge can have devastating effects on the ocean ecosystem, posing risks to marine life and water contamination. By leveraging advanced technology and data analysis, the project seeks to enhance ocean monitoring and ecosystem management practices. The goal is to create sustainable and responsible approaches to protect and preserve the world's oceans and the wildlife that rely on them.

SFI Smart Ocean will consist of three key components combining amphibious hardware, wireless communication and cutting-edge software and middleware[4]. These key components are

- Autonomous battery-powered sensors capable of collecting, partially processing, and transmitting data from sensors in the ocean
- A wireless network connecting and coordinating all these sensors to enable flexibility and multi-direction communication
- A software platform for storing, processing, and analysing all the accumulated data.

As new ocean- and marine data services are emerging, there are challenges related to acquiring sufficient quality for use in smart systems. There is still a huge gap in data coverage with substantial challenges related to interoperability, data- and meta-data standards, and APIs[18].

When working with challenges related to interoperability and communications, models of systems and their communication with the possibility of simulating the interaction and communication is a great advantage. With SFI Smart Ocean there has been presented an initial CPN model [15] of the smart ocean data and application platform. This CPN model is focused on service interaction and brings a high-level of abstraction in the modelling.

1.3 CPN Tools

CPN Tools is a comprehensive software tool designed for editing, simulating and analysing CPNs. Coloured Petri nets are an extension to the Petri net modeling language, which is a mathematical framework for describing distributed systems. CPNs are particularly well-suited for modeling systems that involve communication, synchronization and resource sharing.

The primary goal of CPNs was to create a modeling language that combines theoretical rigor with practical versatility, capable of handling the size and complexity of real-world industrial projects. To achieve this, CPNs integrate the strengths of Petri nets and programming languages. Petri nets provide the primitives necessary to describe the synchronization of concurrent processes, while programming languages provide the tools for defining data types and manipulating data values.

CPN Tools, with its powerful features for editing, simulating, and analyzing CPNs, provides a valuable platform for researchers and practitioners working with distributed systems, communication protocols, embedded systems, automated production systems, workflow analysis, and VLSI chips. The continuous development and improvement of CPN Tools up until now has contributed to advance the understanding and application of Coloured Petri Nets in a wide range of domains.

The development of a modern solution for CPN Tools began with the master thesis [12] as mentioned earlier. The focus of this project was to develop a modernized solution of CPN Tools, with an emphasis on simulation capabilities for .cpn files. The modern solution was predominantly implemented using the F# programming language. The resulting solution operates in a terminal environment and offers simulation functionality, as well as the generation of a state space report for a given CPN model.

1.4 Problem Description

CPN Tools, the most popular existing tool for Coloured Petri Nets, is built using Standard ML[28] and the Beta programming language [29]. However, the development and updating of CPN Tools faces challenges due to the lack of current support for Standard ML and the accumulation of technical debt associated with the use of the BETA programming language for the graphical user interface. Consequently, it becomes difficult to maintain and update CPN Tools in its current state. Recognizing the need for a modernized version of CPN Tools, there is a demand for a continuously maintained and updated CPN editor.

As discussed in Section 1.1, the prevalence of concurrent systems with multiple communication components has increased. For instance, the model presented in [18] exemplifies the necessity for enhanced modularity in models of such complex systems. Therefore, developing a modern CPN editor that addresses these evolving requirements becomes crucial.

The CPN Simulator is already completed and functional, but lacks an accompanying editor to construct models and visualize their simulations in a more user-friendly manner. The editor plays a vital role in facilitating the creation, editing, and visualization of simulations. Its integration with the CPN simulator allows for a comprehensive modeling and simulation environment.

In Chapter 6, an evaluation of the candidate framework solution chosen for the development of the CPN editor will be presented. This evaluation will provide insights into the suitability and effectiveness of the selected framework, shedding light on its capabilities and potential to meet the requirements of a modern CPN editor.

1.5 Research Method

This master's thesis is centered around the outcome of the project, as its progress will be closely related with the progress made on the developed system. The results and achievements of this thesis will be based on extensive research conducted on candidate technologies and the subsequent development of an application we have developed with one of these frameworks.

The research on candidate technologies is based on Brown & Wellnaus "A Framework for Evaluating Software Technology"[3]. This framework serves as a guiding principle for assessing the potential of different software technologies in creating a functional, scalable, and interoperable editor for modern CPN modeling. Furthermore, the chosen framework should possess quality attributes and, to some extent, vendor support. These criteria serve as the focal points for selecting a framework that will be compared to the existing graphical solution provided by CPN Tools.

As depicted in Figure 1.1, the evaluation process began with a descriptive modeling phase where we examined the problem domain and analyzed CPN Tools to identify the necessary features and potential improvements required for a modern CPN modeling tool. Further detail regarding the features of CPN and CPN Tools will be elaborated on in Chapter 2.2.

Subsequently, we moved into the experiment design phase, as it is illustrated in 1.1, where we conducted evaluation on four candidate frameworks. The findings and information related to these frameworkds are presented in Chapter 3.

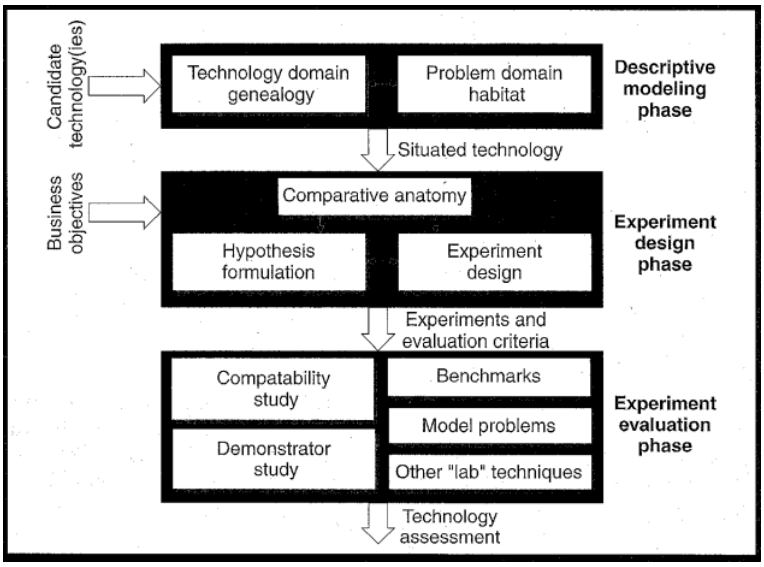


Figure 1.1: Brown and Wellnaus Technology evaluation framework

In this master's thesis, the research methodology entails a functional prototype using a suitable software technology. Following Brown and Wellnau's approach, we identified a fitting framework and evaluated the implementation and editing challenges associated with existing CPN models within the prototype. Moreover, the user experience of the prototype will be compared against the established solution provided by CPN Tools. Additionally, we conducted a comparative analysis with similar software tools, further benchmarking the prototype against CPN Tools.

By employing this research methodology, we aim to contribute to the advancement of CPN modeling tools by delivering a comprehensive evaluation of candidate frameworks and developing a functional prototype that improves upon the existing solution. The comparative analysis will shed light on the strengths and weaknesses of different software technologies and provide valuable insights for the development of future CPN modeling tools.

1.6 Development Method

The adoption of an agile engineering approach has proven to be highly beneficial throughout the project, particularly during the initial phases. By employing sprints and setting specific objectives for each sprint, we have been able to follow a Scrum methodology for sprint management. The duration of each sprint has varied based on the anticipated time required to achieve the defined objectives, ranging from one to four weeks. Additionally, prior to the commencement of each sprint, we have conducted supervisor meeting to discuss and determine the necessary actions and strategies.

The agile workflow was particularly advantageous when it came to selecting a suitable a framework using Brown and Wellnau's approach. This approach emphasizes the evaluation of various quality attributes of software technologies, including performance, scalability, maintainability, and interoperability. By adopting an agile workflow we were able to closely monitor the project's progress and ensure that the chosen framework aligned with our specific requirements and objectives.

Throughout the project, we extensively investigated several potential technologies, including Eclipse EMF Core, Eclipse Graphical Language Protocol, Elm, and GoJS. However, after careful consideration and evaluation, we ultimately arrived with a technology stack that includes Electron, GitHub, GoJS, JavaScript, HTML, and CSS. Specifically, we leveraged Electron, a framework for building desktop applications using JavaScript, HTML, and CSS, to develop the prototype of our CPN editor. The codebase was securely stored on GitHub, and we utilized their issue board functionality to effectively track and manage tasks and goals. Furthermore, GoJS, a powerful modeling framework, played a crucial role in creating the palette and modeling components of our editor. By utilizing JavaScript, HTML, and CSS we were able to develop and customize the application, harnessing the capabilities of Chromium to deliver the desired functionality and user experience.

1.7 Research Questions

Having an already modern and existing compiler and runtime environment for coloured Petri net model guides our focus to the graphical part of the tool. Our research aims to address the following key questions that will drive the development of a modern take on the graphical editor.

What specific functionalities are essential for a CPN editor to effectively support modeling and simulation activities? In order to provide a comprehensive tool for CPN modeling, it is crucial to identify the core functionalities required to facilitate the creation, editing, and simulation of CPN models. By exploring the needs when using CPN Tools and considering the CPN tools, we aim to determine the essential features that enhance the modeling and simulation experience.

What are the strengths and limitations of various candidate software technologies in meeting the identified requirements? We will evaluate and compare different software technologies, such as GoJS, Eclipse GLSP and other potential frameworks, to assess their suitability for building a modern CPN editor. By examining their capabilities, integration potential, and ease of use, we can identify the most appropriate technology to support the graphical modeling aspects of CPN.

To what extent does our developed prototype demonstrate the necessary functionality and usability required for an effective CPN editor, as identified earlier? Through the development of a prototype CPN editor, we will strive to create a tool that meets the identified requirements. By implementing the essential functionalities and evaluating the usability of the editor, we aim to validate its effectiveness in supporting CPN modeling and simulation tasks.

- RQ1 What specific functionalities are essential for a CPN editor to effectively support modeling and simulation activites?
- RQ2 What are the strengths and limitations of various candidate software technologies in meeting the requirements identified in RQ1?
- RQ3 To what extent does our developed prototype demonstrate the necessary functionality and usability required for an effective CPN editor, as identified in RQ1?

By addressing these questions, we seek to contribute to the advancement of graphical modeling tools in the context of CPN, ultimately providing a more intuitive and efficient solution for CPN practitioners

1.8 Outline

The section provides an overview of the contents of the next chapters. We refer to the CPN editor environment developed in this thesis as both the application and the editor.

Chapter 2 provides background information about the two-phase commit protocol, Coloured Petri Nets and its functionalities. We enumerate functionality required for a modern coloured Petri net editor. Understanding the basics of features in coloured Petri nets and the CPN Tools software is important when deciding which features to implement and prioritize when developing a new framework. We use the two-phase commit protocol as example for the different features in CPNs and CPN Tools.

We discuss the candidate software technology platforms in Chapter 3. We have done some experimenting with these software technologies to asses which is the most appropriate to develop the new CPN editor. These technologies are Eclipse EMF Core, Eclipse Graphical Language Protocol, Elm lang, GoJS and ElectronJS. It was critical that the framework we chose was modern and with continuous support for lasting as a foundation for our application. Lastly we provide a summary of how appropriate the technology is for the development.

Chapter 4 introduces the design and implementation of the CPN editor environment we have developed. This includes explanation of the implementation of the different components and features of the new application. The features we included are based on the requirements we set out in Chapter 2. Some of the solutions are based on challenges encountered and shows the most suitable solution we found at the time.

Chapter 5 is an explanation of the code structure we ended up with when developing the application. Here we explain the relationship between the components to provide an overview of how the application is structured.

In Chapter 6, we evaluate the new CPN editor environment. We deliver a step-by-step explanation of how to create a CPN model of two-phase commit protocol in the application we have developed. We compare the features of our application to the corresponding features in CPN Tools.

Chapter 7 serves as the conclusion of our implementation, where we discuss the

overall outcomes of our project. We also discuss related work, and future work on our application that can be improved upon from its current state.

In this thesis we discuss coloured Petri nets, mostly through CPN Tools. It is not assumed the reader has any prior knowledge of either of these as we provide detailed explanation of these. We do this to give an understanding of how important the modernization of editing CPNs is. We will also discuss JavaScript and some HTML, but we assume the reader has basic knowledge about these topics.

Chapter 2

Background

This chapter provides an in-depth background on CPN Tools, a widely used software tool for modeling and simulating Coloured Petri Net (CPN). We will explore the features offered by CPN Tools and identify the key aspects that are essential for a new CPN editor. By understanding the strengths and limitations of CPN Tools and considering the evolving needs of CPN practitioners, we can lay the foundation for designing and developing a modern CPN editor that caters to their requirements.

To gain insights into the necessary functionality for an effective CPN editor, we will use the two-phase commit protocol as an illustrative model throughout this chapter. The two-phase commit protocol is a widely studied and commonly used protocol for distributed transactions. By examining how CPN Tools handles the modeling and simulation of this protocol, we can identify the specific features and functionality that are essential for an effective CPN editor.

2.1 Two-phase commit protocol

To illustrate the concepts and capabilities of Coloured Petri Net (CPN) and their application in CPN Tools, we will examine a two-phase commit protocol as an example model. The two-phase commit protocol is well known protocol used in distributed systems to ensure the consistency of transactions across multiple participants.

Figure 2.1 presents a graphical representation of the two-phase commit protocol model in CPN Tools. This model showcases various features and elements of CPNs, providing insights into how CPN Tools can be used to describe and analyze distributed systems.

The two-phase commit protocol model demonstrates the utilization of places and transitions, which are fundamental components of Petri nets. Places depicted as circles and transitions represented as rectangles. Due to this being a CPN, tokens are denoted by markings, to indicate the state of the system at a given moment.

By examining the two-phase commit protocol model in CPN Tools, we can delve

into the specific functionalities and capabilities of the tool. This includes the ability to define and specify communication and synchronization mechanisms, simulate transactions, and verify the correctness of the protocols execution by using the functionality of CPN Tools.

We will now explain the behaviour of the two-phase commit protocol.

The two-phase commit protocol is a type of atomic commitment protocol. It is a distributed algorithm that coordinates all the processes in a transaction program, and decides between committing or rolling back a transaction [21]. The basic algorithm of the two-phase commit protocol consists of two phases: the commit request and the commit phase.

The first phase is the commit request phase. This starts by the coordinator sending a query to all participating workers and waits until it receives a reply from all of participating workers. Participants will then execute their transaction up until the point where they will commit. Workers will then send a reply if they will commit or not, a Yes to commit or No to abort the commit. If there is a single abort, the protocol will not allow committing.

In the commit phase there is a possibility for a success, (which is a commit), or a failure, (which is aborting). In case of success, the coordinator receives Yes messages from all participating workers. The Coordinator replies with a commit message to all workers, where they will complete this operation. Workers will then send back an acknowledgement to the coordinator. The coordinator completes transaction when all acknowledgements has been received.

In case of failure, the coordinator will send an abort message to all workers. Workers will then go back to their initial state and send an acknowledge message to coordinator. The coordinator undoes the transaction when all acknowledgements has been received.

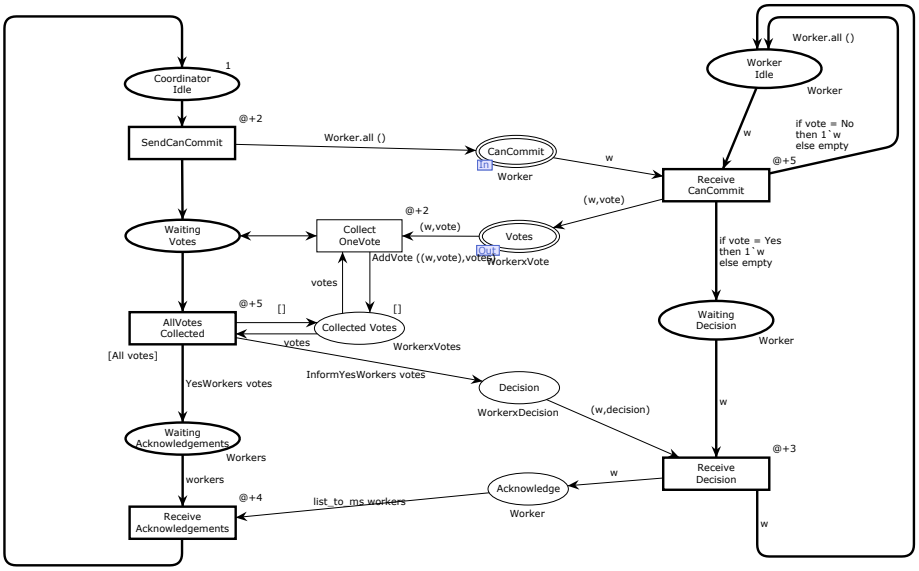


Figure 2.1: Two-Phase Commit Protocol in CPN Tools

In the following Section, we introduce several features introduced in Figure 2.1 to demonstrate the capabilities of CPN Tools. These features are essential for effectively model and analyzing distributed systems using coloured Petri nets. By exploring these features, we aim to highlight the power and versatility of CPN Tools as a graphical modeling tool, and find the functionality needed for developing a modernization for the modeling capabilities.

2.2 Coloured Petri Nets

In this chapter we look into the diverse set of features in Coloured Petri Nets and CPN Tools. The features in CPN Tools play a crucial role in enhancing the modeling experience and enabling users to effectively analyze and validate their models. Each feature contributes to different aspects of the modeling process, from creation and editing of models to the simulation and analysis, though in this chapter we will mainly focus on creation and editing.

2.2.1 Places and transitions

In CPNs, places and transitions are fundamental components for modeling systems. Places are represented by ellipses, while transitions are represented by rectangles. They serve different purposes in capturing the behaviour of the system.

Places in CPN Tools are used to represent the state of the system. They can hold tokens, which will be explained in Section 2.2.4. Places are essential for modeling the state variables and tracking the progress of the system.

Transitions, on the other hand, model the actions or events that can occur in the system. They are responsible for the state changes and the flow of tokens between places. Transitions can be enabled or disabled based on the availability of tokens in their input places. Input places are places connected to the transition through arcs. An enabled transition means that it has the necessary tokens in its input places to fire or execute.

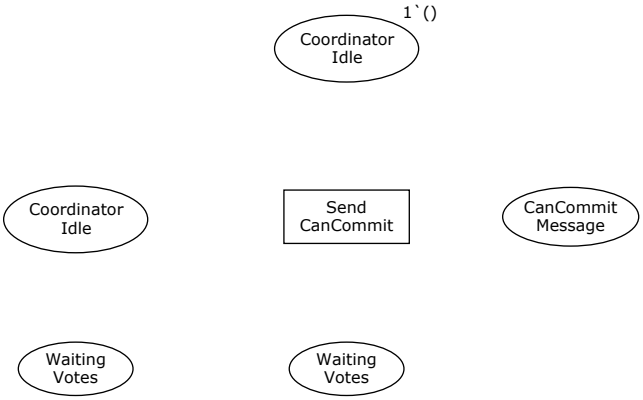
When an enabled transition fires, it removes one or more tokens from each of its input places, this is also called consuming tokens. It will then adds tokens to each output place, also called producing tokens. This represents the execution of an action or an event in the system. The firing of a transition can trigger state changes and cause the system to move from one state to another.

In CPN Tools, transitions can also have conflicts and priorities. Conflicts occur when multiple transitions are enabled simultaneously, and only one can be fired at a time. Priorities can be assigned to transitions to determine which transition should be given preference in case of conflicts. Transitions with higher priority levels are selected for firing over transitions with lower priority levels.

By using places and transitions in CPN Tools, one can effectively model the state and actions of a system. The interaction between places and transitions, along with conflicts and priorities, allows for the representation of complex behaviours and decision-making processes within the system.

In figure 2.2 we can see part of the coordinator. `Coordinator Idle` is the

starting place of the protocol. Waiting votes is the coordinator state waiting for worker response. In Figure 2.2b the first transition comes in, this will start sending messages asking workers if they can commit. `CanCommit Message` is a state where coordinator has constructed the message to workers, asking for their own state of readiness to commit.



(a) Places (b) Transition with three places

Figure 2.2: Examples of places and transition.

2.2.2 Colour sets

The colour sets in CPN Tools are used to define the data types that can be assigned to tokens in places. CPN ML, the programming lanugage used in CPN Tools based on Standard ML, provides several base colour sets that represent common data types. These base colour sets include:

- `UNIT`: Represents a unit type with a single value.
- `INT`: Represents integer values.
- `STRING`: Represents string of characters.
- `BOOL`: Represents boolean values(true or false).
- `REAL`: Represents real numbers(floating-point values).

These base colour sets allow for the definition of tokens with different data types and values in CPN models.

Additionally, colour sets can be more structured by combining existing colour sets or defining new ones. In the CPN ML code provided in Listing 2.1, lines 2-8 demonstrate the use of a structured colour set. The `WorkerxVotes` colour set is defined by combining the `Worker` from line 2 with the `Vote` colour set from line 4. This allows for tokens in places of the `WorkerxVotes` colour set to hold data consisting of both a worker identifier and a vote value.

By defining and using structured colour sets, CPN Tools provides flexibility in modeling complex data types and relationships within a CPN, such as the two-phase commit protocol as these colour sets are based on.

```
// Types
colset Worker = index wrk with 1..W;
colset Workers = list Worker;
colset Vote = with Yes | No;
colset WorkerxVote = product Worker * Vote;
colset WorkerxVotes = list WorkerxVote;
colset Decision = with abort | commit;
colset WorkerxDecision = product Worker * Decision;

// Variables
var w: Worker;
var workers: Workers;
var vote: Vote;
var votes: WorkerxVotes;
var decision: Decision;

// Values
val W = 2;
```

Listing 2.1: CPN ML definitions for two-phase commit protocol

2.2.3 Marking and multi-sets

In CPN and CPN Tools, a multi set is used to represent the contents of a place. A multi-set is a collection of tokens that allows for multiple tokens with the same value. It is defined as a function from a domain(in this case, the colour set of a place) into the set of natural numbers[16]. Each token in the multi-set is associated with a specific value from the colour set, and the number associated with each value indicates the multiplicity or count of that token in the multi-set.

Operations on multi-sets include addition(`++`), subtraction(`--`), and comparison (`<<=`). The addition operation combines two multi-sets, merging the tokens from both sets. The subtraction operation removes tokens from a multi-set based on another multi-set. The comparison operation checks if one multi-set is a subset of or equal to another multi-set.

In the context of the two-phase commit protocol example shown from in Figure 2.1, the `Idle` place holds a multi-set of tokens. In this case, the tokens in the multi-set are instances of `wrk(1)` and `wrk(2)`, with counts of 1 for each token. The addition(`++`) notation represents the union operation, indicating that the multi-set contains both tokens.

This use of multi-sets allows CPN Tools to model and represent the presence of multiple tokens with the same value within a single place, capturing the notion of multiplicity or repetition in the system being modeled.

2.2.4 Tokens and current marking

In CPN and CPN Tools, tokens are the individual elements that are held by places. A place can hold multiple tokens, each representing a specific value from the colour set of the place. The marking, on the other hand, is a distribution of tokens across the places in the CPN model, representing the state of the system.

In Figure 2.3, the places `CoordinatorIdle` and `WorkerIdle` are shown with 1 token each. The green circle on the place indicates the number of tokens present,

while the marking is represented as `1‘()`, where the 1 is the count of the specific value `()`. In this case, both `CoordinatorIdle` and `WorkerIdle` have one token each, and the value of each token is `()`.

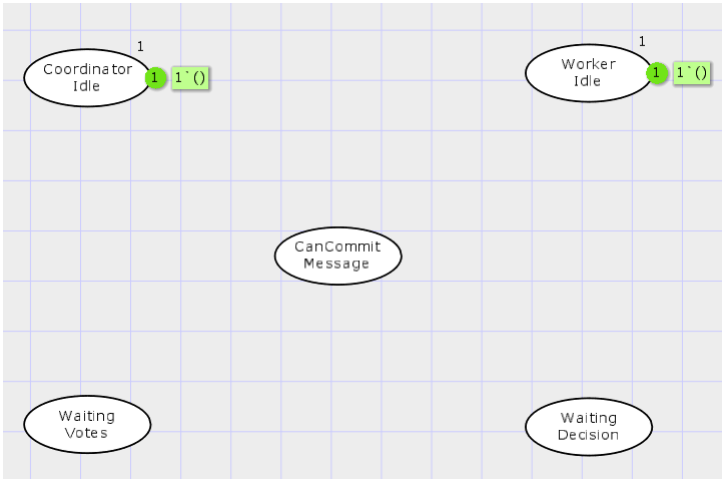


Figure 2.3: First phase of 2PC simplified and without transitions

The initial marking represents the initial state of the system. It is defined by specifying the tokens that are initially present on the places. In the CPN model, the initial marking is indicated by the token initially placed on the "starting" places, typically specified using an expression in the top right corner of the place.

As the CPN model is simulated or executed, the marking representing the current state of the system changes. Initially, the current marking is the same as the initial marking. However, as the simulation progresses and transitions are fired, tokens may be removed from some places and added to others. causing the current marking to change accordingly.

The markings and tokens provide a visualization and tracking the state of the system throughout the simulation or execution process. They help capture the dynamic behavior of the system and allow for analysis and understanding of its state transitioning.

2.2.5 Arcs and arc weights

Arcs in CPN Tools are essential components used to connect places and transitions within coloured Petri nets models. They represent the flow of tokens between these elements and play a crucial role in defining behaviour and dynamics of the modeled systems. Arcs can contain arc inscriptions or expressions which is CPN ML code snippets that may be used to alter the path of a token in a system. These inscriptions provide flexibility and allow for dynamic behavior within the model. Additionally, the presence and properties of arcs determine the enabling and occurrence of transitions in a model.

Normal arcs are the most basic and commonly used type of arcs in CPN Tools. They facilitate the transfer of tokens between places and transitions. Normal

arcs can carry a single token or a weight, indicating the number of tokens being transferred. These arcs define the flow of control in the CPN model and are essential for modeling the sequence of activities. The type of an arc expression must match the colour set of the place connected to the arc, ensuring compatibility and consistency in the model.

Arc weights are used to specify the number of tokens required for enabling a transition, as well as the number of tokens consumed and produced during its occurrence. By assigning appropriate weights to arcs, the modeler can control the conditions for transition firing and accurately represent the resource requirements of the system.

Alternative arcs

CPN Tools offers several types of arcs, each serving a specific purpose and providing unique functionalities. These are the most common arcs used as extension to the default arc.

Inhibitor arcs introduce a form of control to the token flow in CPN Tools. They are used to specify conditions under which a transition is inhibited or prevented from firing, even if all other input places have the required number of tokens, Inhibitor arcs are represented with a small circle on the arc, and their purpose is to restrict the firing of transitions based on specific conditions.

Reset arcs allow the removal of tokens from places without consuming them. This means that when a token traverses a reset arc, it triggers the removal of tokens from the connected places without affecting the token count or state of the transition. They are used to reset or clear the tokens in a place when a specific condition is met. Reset arcs are particularly useful when modeling systems that require the resetting of certain variables or states during the execution of transitions. Reset arcs are represented by a double arrowhead.

Figure 2.4 represents a simplification of the first phase in the two-phase commit protocol. The coordinator is depicted as a single token to represent its state, enabling the `SendCanCommit` transition. On the other hand, the `Worker Idle` place also has a token, but this token cannot be consumed by any transition because the transitions require tokens from all input places to become enabled. Consequently, the `ReceiveCanCommit` transition remains disabled as it relies on both `Worker Idle` and `CanCommit Message` tokens.

During the execution of the first stage, a token from `Coordinator Idle` place triggers the enabling of the `SendCanCommit` transition. When this transition fires, it outputs a token to both the `WaitingVotes` and `CanCommitWorker` places. As a result, The `SendCanCommit` transition becomes disabled, and the `ReceiveCanCommit` transition becomes enabled. Moving on to the second stage, the `WaitingVote` place will continue waiting, since there are no enabled transitions to execute. Meanwhile, the `ReceiveCanCommit` transition transfers its input tokens to the `WaitingDecision` place and becomes disabled.

The simplified representation illustrates the flow of tokens and the enabling/disabling of transitions in the first phase of the two-phase commit protocol. It captures the progression of the system state as tokens are moved between places

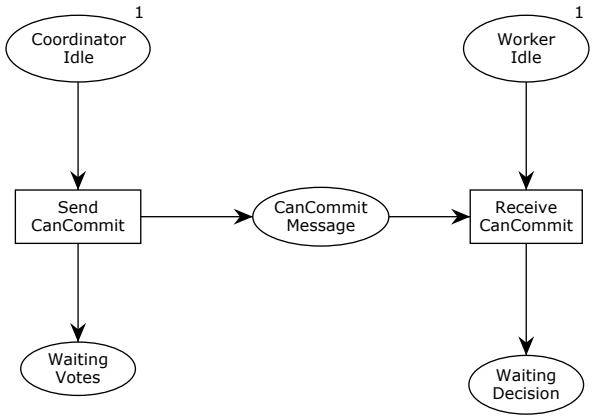


Figure 2.4: Arcs connecting places and transitions

and transitions are enabled or disabled based on the availability of tokens in the input places.

2.2.6 Transition variables

In Figure 2.5, the arcs leading into the `Receive CanCommit` transition contain arc expressions that involve free variables. Free variables are output arc variables that have not been bound by an input arc or in a guard. In this case, the free transition variables are `w` and `vote` going from `Receive CanCommit` transition to the `Votes` place.

For transitions to be enabled or occur, variables must be bound to values. This is similar to parameters known from programming. The association of values to variables is called a transition binding. The binding corresponds to the possible enabling and occurrence modes of the transition. Not all possible bindings will in general be enabled. The scope of a variable is the surrounding arc expressions of the transition.

In the first stage of the worker section depicted in Figure 2.5, the workers are idling and waiting for the coordinator to send request regarding their commit status. When the `CanCommit` and `Worker Idle` places each have two tokens `1‘wrk(1)++1‘wrk(2)`, indicating the presence of one worker 1 and one worker 2, the `ReceiveCanCommit` transition becomes enabled due to all input places having tokens.

In This part of the model, the `ReceiveCanCommit` transition sends the worker name along with their vote to the `Vote` place. Depending on the vote, the worker will either be moved to the `Waiting Decision` place if the respective worker's vote is yes, or back to the `Worker Idle` place if the vote is No.

The use of transition variables in arc expressions allows for dynamic evaluation and flexible behavior within the CPN models. It enabled the model to capture different scenarios and transitions based on the values associated with the transition variables, contributing to the versatility of the model in representing complex system dynamics.

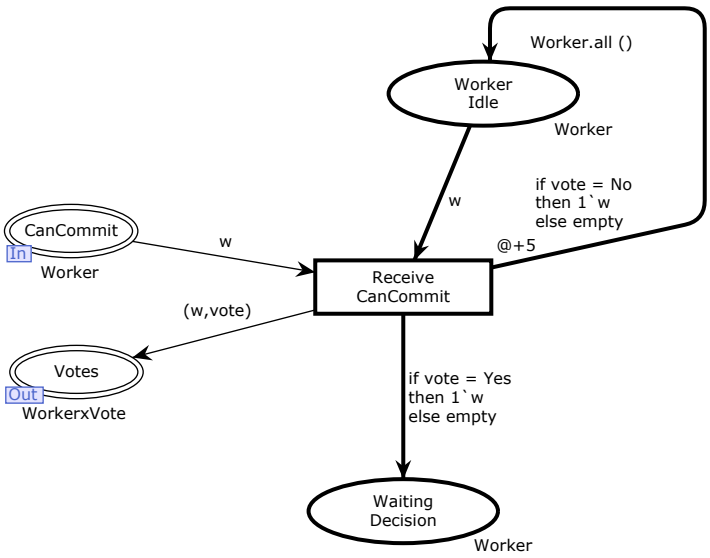


Figure 2.5: ReceiveCanCommit has the free variables; w and vote

2.2.7 Guard expressions

The inclusion of boolean guard expression in transitions enhances the flexibility and control over the behavior of the system in coloured Petri nets. These guard expressions serve as additional enabling conditions, allowing us to impose restrictions on when a transition can be triggered.

In the context of two-phase commit protocol, we can observe the use of guard expressions on the `AllVotes Collected` transition. This transition is associated with the guard expression `All votes`. This ensures that the transition will only be enabled if all the tokens containing votes are present in its input place, which is `Collected votes`.

By utilizing the guard expression, we can enforce the condition that all the votes must be collected before the `AllVotes Collected` transition can occur. This allows us to model the behaviour of the system accurately, ensuring that the necessary conditions are met before progressing to the next phase or action.

The ability to incorporate guard expressions in transitions adds an additional

layer of control and specificity to the behavior of the system, enabling us to capture complex conditions and constraints within the CPN model.

2.2.8 Substitution transitions

Substitution transtions provide a powerful mechanism for structuring and abstracting complex models in CPNs. With substitution transitions, a transition can represent an entire piece of net structure, allowing for encapsulation and modularity.

When a substitution transition is used, the net structure contained within it remains physically present and executable. However, it provides a higher-level view of the model, hiding the internal details and complexity. This abstraction improves the readability and manageability of the model, making it easier to understand and work with.

In the context of supermodules and submodule, a module that contains a substitution transition is considered a supermodule relative to the submodules it may contain. The supermodule encapsulates and abstracts the functionality represented by the substitution transition, providing the higher-level perspective.

Submodules refer to the contained logic within the substitution transition. These submoduels can be opened and examined to understand the detailed internal structure. They allow for a deeper understanding of the specific functionality represented by the substitution transition.

Figure 2.6 demonstrates an example of a supermodule containing a substition transition. By examining the contained submodule in figure 2.7, we can gain insight into the detailed logic and behavior encapsulated within the substitution transition.

Overall, substitution transitions offer a powerful means of structuring and abstracting complex models, enabling better organization, modularity, and understandability in CPNs.

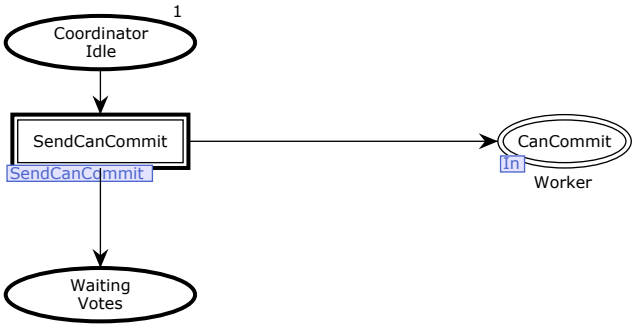


Figure 2.6: Substitution transition SendCanCommit

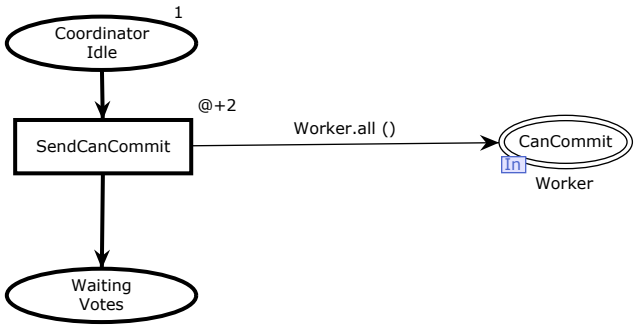


Figure 2.7: Submodule of substitution transition SendCanCommit

2.3 Requirements

In order for the modernization of the existing CPN Tools to be useful, certain requirements should be considered. Here are some key requirements related to the technology platform.

Continued support and active development: The chosen technology platform should be actively supported and maintained to ensure long-term viability. This includes regular updates, bug fixes, and compatibility with modern operating systems.

Well-documented: The technology platform should have comprehensive and up-to-date documentation that is easy to understand and navigate. This documentation should cover both basic usage and advanced features, providing users with a clear understanding of the platforms capabilities.

User-friendly interface: The modernized CPN editor should have an intuitive and user-friendly interface. It should be designed to facilitate efficient modeling, simulation, and analysis of coloured Petri nets. The interface should prioritize ease of use and provide a smooth learning curve for new users. Having a framework supporting this is of great interest.

Rich feature set: The framework should encompass a wide range of features and functionalities to support modeling Coloured Petri nets. This includes functionality like copy, cut, paste, undo, and redo, which are crucial for manipulating and organizing Coloured Petri nets objects within the tool. Additionally, the presence of supplementary functionalities implemented by default will be considered as valuable enhancements, contributing to the overall functionality and usability of the modeling tool.

Compatibility and interoperability: The modernized CPN editor should be compatible with standard file formats, ensuring seamless collaboration with other tools and systems.

Community and support: The technology platform should have an active community of users and developers who can provide support, share knowledge, and contribute to the tools improvement. This includes forums, discussion groups, and online resources for users to seek assistance and exchange ideas.

A new software would need some of the features explained in Section 2.1 to function as an editor.

These parts includes a modelling canvas, used for creating and editing a model. Support for nodes in the form of places, transitions, and substitution transitions. The different nodes also need their functionality. For places the following fields are needed; place name, marking, color set, port and socket field. Transitions need the following fields; transition name, guard expression, time inscription, CPN ML field for inscriptions and a priority field. Substitution transitions need some sort of linking to their sub models. Connecting the nodes, the software will need arcs with inscription fields for arc inscriptions, and possibilities for some alternative arcs; Inhibitor Arcs and Reset Arcs.

The following section will discuss these in detail.

2.3.1 Required functionality

Places

A CPN model require places to indicate the state of the a model. The following specifications should be implemented for complete functionality in the modernized CPN Tools:

Places should have inscription fields to specify the initial marking and colour set. The initial marking inscription field allows users to define the initial distribution of tokens on the place. The colour set inscription field specifies the type or domain of tokens that can reside on the place.

There should be a included a field to display the current marking of the place. This field dynamically updates to show the current state of the place, indicating the number and distribution of tokens.

Transitions

In CPN models, transitions play a crucial role in representing the actions and events within a model. To effectively capture the behaviour of transitions, it is important to include the following connected input fields:

- Guard Expression - Transitions may have boolean guard expressions associated with them, as an additional enabling condition. Guard expressions restricts the firing of the transition based on a certain conditions or values in the model.
- Action field - This field represents the action or code segment that is executed when the transition occurs. It defines the specific behavior or functionality associated with the transition.
- Priority - Transitions can be assigned a priority value, which determines their relative importance or order of execution in the model. This allows for the management of concurrent or conflicting transitions.
- Time inscription - Time inscriptions are used to simulate the duration or time-related aspects of a specific task or action. They specify the amount of time required for the transition to complete its execution

Furthermore, it is beneficial to have a mechanism to highlight transitions to indicate their enabled state. By highlighting transitions, users can easily identify which transitions are currently enabled and available for firing in the model, This visual feedback provides a clear understanding of the model's behaviour and facilitates effective analysis and simulation.

By incorporating these connected input fields and enabling visual highlighting, the modeling tool ensures comprehensive representation and interaction with transitions in CPN models.

Markings

Markings in CPN play a crucial role in representing the state of places. As described in Section 2.2.3, they serve as indicators of the current state of a place within a Coloured Petri net. By visually representing the number of tokens or elements contained within a place, markings provide essential information about the system's state and behavior.

Arcs

To establish the relationship between places and transitions in CPN models, we require arcs for connecting them. Arcs serve as the bridges between places and transition, enabling the flow of tokens and influencing the behavior of the system.

Similar to places and transitions, arcs also possess an inscription field, which allows us to specify arc inscriptions. This inscription field will provide additional information or conditions that affect the flow of token along the arcs. By defining arc inscriptions, we can control the interaction between places and transitions, influencing when and how transitions are enabled or fired based on the state of the connected places.

In addition to standard arcs, we should implement the two other types of arcs that are important to consider: inhibitor arcs and reset arcs. Inhibitor arcs restricting the firing of a transition by checking if a specific condition is not satisfied in the connected place. Reset arcs remove tokens from the connected place when the associated transition is fired.

In our modernized CPN tool, it is essential to provide support for all three types of arcs: standard arcs, inhibitor arcs, and reset arcs. Users should be able to define and switch between these arc types based on the modeling requirements. By offering this flexibility, we can accurately capture the dynamics and behaviour of the system being modeled.

Substitution transitions

To support the representation of larger net structures and enable abstraction, our modernized CPN tool must include the capability to handle substitution transitions. Substitution transitions allow us to encapsulate complex net structures within a single transition making the model more modular and manageable. This means including the ability to create submodules, and supermodules, enabling hierarchical organization and composition of the model.

While simulation on substitution transitions is not a top priority, it would be a valuable bonus feature to have. Simulating on substitution transitions would provide a more complete and accurate understanding of the system's behaviour, allowing users to observe the dynamics within the abstracted components.

To effectively work with abstraction, it is crucial to have a clear and intuitive overview of the hierarchical structure of the models. A visual representation, such as a treeview or a hierarchical map, would greatly aid in comprehending and navigating the abstraction hierarchy. This visual representation would display the relationship between submodules and supermodules, providing users with a high-level view of the abstracted components and their interconnections. This feature would enhance the usability and maintainability of the models making it easier to work with complex and abstracted systems.

Runtime environment

When developing a modeling tool, one of the crucial decisions is to determine the platform on which it will run. There are two potential alternatives to consider: an independent software application or a browser-based tool. This choice will determine where the models are created and manipulated, and it has significant implications for the tool's accessibility and usability.

Regardless of the chosen platform, it is essential to provide the capability to create nodes of different types and establish connections between them. This functionality allows users to define and represent various components or entities within their models. Users should be able to easily create, edit, and delete nodes, as well as establish and modify the links between them. This flexibility enables users to refine and adjust the relationships as needed.

In addition to node creation and connection, integrating a functional code field into the modeling tool is a valuable feature. This allows users to incorporate functional code snippets or logic directly within their models. By providing a code field, users can implement custom behaviors or algorithms, enhancing the expressiveness and flexibility of their models. This functionality is particularly beneficial for capturing complex behaviors and interactions that go beyond the visual representation of the nodes and connections.

By incorporating these features, the modeling tool offers a versatile and robust environment for creating, editing, integrating elements within the models. It provides the users with the necessary tools to express their ideas effectively, manipulate the models structure, and incorporate custom behaviors. This comprehensive approach enhances the overall modeling experience, making the tool a powerful resource for system analysis and design.

Saving and loading

To ensure a seamless user experience, it is important to provide the functionality for saving and loading models in the modeling tool. This allows users to save their work and resume it at a later time, ensuring that no progress is lost. Additionally, it enables users to share their models with others who are using the same software, allowing collaboration and knowledge exchange.

When implementing the saving and loading functionality, one important requirement is to use a file format that is relatively easy to read for users. This ensures that users can easily understand and work with the saved files, even outside of the modeling tool. A common file format that meets this requirement is JSON (JavaScript Object Notation). JSON is a lightweight data interchange format that is human-readable and widely supported across different programming languages and platforms. Using JSON for saving and loading models makes it easy for users to share their models with others and allows for interoperability.

By utilizing JSON for saving and loading models, users can easily comprehend and work with the saved files. JSON files can be shared, transferred, and opened in different software applications, fostering interoperability, collaboration, and seamless integration into workflows. This user-friendly approach empowers users to effectively manage their models, enhance collaboration, facilitate knowledge sharing, and work with models both within and outside of the software environment.

Modularity

In our modernization effort, one of our primary goals is to introduce a higher degree of modularity to the CPN editor. We plan to achieve this by separating submodules within substitution transitions into individual files, which offers several benefits and improvements.

Breaking down a model into individual files for submodules in substitution transition helps in organizing the codebase more effectively. Each submodule can have its own dedicated file, containing the relevant logic, functionality, and design elements. This modular approach enhances maintainability and makes it easier to navigate and understand the overall code and model structure.

Modularity allows for better reusability in code. Submodules within substitution transitions can be designed as self-contained modules that can be reused in multiple instances within the editor. This promotes code efficiency and reduces duplication, as the same submodule logic can be utilized across different transitions or even different models.

The use of individual files for submodules simplifies the development and testing process. Developers can focus on individual submodules without being burdened by the complexity of the entire editor. This modular approach facilitates easier debugging and integration testing of specific submodule functionalities, leading to faster development cycles and improves software quality.

A modular CPN editor allows for greater flexibility and customization capabilities. Different submodules can be developed independently, offering the ability to tailor the behaviour and appearance of each submodule to specific requirements. This flexibility enables users to create unique CPN models by combining and configuring various submodules according to their specific needs.

With individual files for submodules, collaboration becomes more seamless. Multiple developers can work on different submodules simultaneously without conflicts, as each submodule is independent and can be developed separately. This modular structure also enables team members to work on different parts of

the editor concurrently, accelerating the development process and allows more collaboration.

A modular CPN editor is highly scalable and extensible. As additional submodules are implemented, they can be seamlessly integrated into an existing system model without disrupting the functionality of other components in the model. This makes it easier to introduce updates, enhancements, or new modules to meet evolving user requirements and accommodate future growth.

Overall, the introduction of modularity through individual files for submodules brings significant improvements to the CPN editor, including better organization, code reusability, simplified development and testing, enhancing customization, streamlined collaboration, and scalability for future expansions.

Common editing operations

The common editing operations, such as cut, copy, paste, undo and redo, are essential features in many software applications, including text editors, graphic design tools, and even productivity software like word processors and spreadsheets. These operations serve several important purposes:

- The **cut** operation allows users to remove a selected portion of text or content and place it into the clipboard. This operation is useful for moving or relocating content within a document or between different documents. It effectively removes the selected content from its original location and prepares it for insertions somewhere else.
- **Copying** enables users to duplicate selected content without removing it from its original location. The copied content is placed into the clipboard, allowing users to insert it multiple times into different parts of the document or into other documents. Copying is particularly valuable when users need to create duplicates or repetitions of certain elements.
- The **paste** operation is used to insert content from the clipboard into the document at the current cursor position. It allows users to place previously cut or copied content into a desired location. Paste is often used in conjunction with cut or copy to transfer content within or between documents.
- **Undo** is crucial feature that allows users to revert the most recent action or series of actions. It provides a way to reverse changes or mistakes, restoring the document to a previous state. Undo is particularly helpful when users make errors or need to backtrack and undo a series of actions.
- **Redo** complements the undo operation by allowing users reapply actions that were previously undone. It is useful when users change their mind after undoing an action and want to revert the revert, effectively restoring the document to a state after an undo operation

It would be a great advantage to find a framework where these common editing operations are implemented as default. So that there is not needed to allocate much time developing this, as there has already been developed for almost all editing tools.

2.3.2 Summary

The following table summarizes the required features for developing a functional CPN editor. In the future chapters we will refer to the listed requirements using the identification in Table 2.1.

ID	Name	Summary
R1	Places	For indicating the initial and current state of CPN models.
R2	Transition	Representing actions and events within a model.
R3	Markings	Indicating the state of a place.
R4	Arcs	Connecting places and transition, indicating the flow of tokens in the model.
R5	Substitution transitions	Representation of larger pieces of a model, introduces abstraction.
R6	Runtime environment	Modelling canvas, palette, treeview, buttons with extra functionality.
R7	Saving and loading	Saving and loading to and from a file..
R8	Modularity	Have substitution transitions in individual files.
R9	Common editing operations	Support for operations such as cut, copy, paste, undo and redo.

Table 2.1: Requirements for a modular CPN editor

These requirements serve as the foundation for developing the CPN Editor and will be referenced in the subsequent chapters to ensure the implementation meets the desired functionalities.

Chapter 3

Software Technology Platforms

In this chapter we survey and evaluate potential frameworks that have the capabilities to fulfill the requirements listed in the previous chapter for developing a modular CPN editor.

3.1 Candidate software platforms

The evaluation of these frameworks involves an investigation into their features, documentation, community support, and suitability for the development of a CPN editor. Our goal is to identify a framework that meets the requirements outlined in the previous chapter, and also offers the necessary tools and capabilities to create a robust and user-friendly modeling tool.

To evaluate the suitability of the different frameworks, we have conducted "get started" tutorials for each of them, which allowed us to gain a firsthand understanding of their capabilities and determine their viability for our project. For the frameworks that showed promise, we proceeded to implement a simple two-phase commit protocol as a practical test to further evaluate the frameworks. Through these evaluations, we gained valuable insights into the strengths and limitations of each framework. In the following sections, we introduce the tested frameworks and provide explanation of the result of the conducted tests. This analysis assist us in making an informed decision regarding the most suitable framework for developing our modular CPN editor.

3.2 Eclipse EMF Core

During the early stages of the evaluation process, we looked into the Eclipse EMF Core framework[8]. However, after completing some initial tasks and considering its usability, we decided not to proceed with further testing due to the framework occasionally being cumbersome to work with.

According to some software developers, the Eclipse EMF framework may be considered overly complex for our intended functionality. While it is an established framework, it is also relatively old, which could be viewed as both a positive and a negative factor. One potential benefit of its age is that it has had time to establish itself and its usefulness, whereas some newer frameworks may be more experimental and untested. However, some developers feel that the outdated nature of the framework can give it a feeling of being obsolete, particularly when compared to newer, more modern alternatives. We did not find any data on the feature set of Eclipse EMF Core.

It is important to note that opinions on the Eclipse EMF framework can vary widely among developers and organizations, and the suitability of the framework may depend heavily on specific project needs and requirements. Additionally, it is worth considering other factors beyond the framework itself, such as the availability of skilled developers and community support for the framework.

3.3 Eclipse Graphical Language Protocol

The Graphical Langugage Server Platform (GLSP)[13] is a versatile framework that supports the development of diagram editors on various Integrated Development Environments (IDEs), including Visual Studio Code, Eclipse Theia, and Eclipse desktop application. The GLSP architecture promotes flexibility by facilitating clear separation of domain-specific diagram logic from the rendering part. This framework allows customization and extensibility of the diagram client by adding custom shapes or editing features through the defined protocol, which can be extended with custom messages if necessary. GLSP provides several implementation options, including servers written in either Java or Typescript based on NodeJS, and source model management based on any format or framework. Additionally, GLSP editors can be integrated into any web application, and dedicated integration components are provided for deployment inside of supported IDEs.

The GLSP framework is designed with a strong focus on customizability and extensibility. This is achieved through the use of two principles: dependency injection and slim abstraction with direct access to the underlying technology. Dependency injection enables every service and component to be configured in a global DI container, providing adopters with the same power for their diagram editors as the framework authors. Slim abstraction and direct access to the underlying technology provide full control over the rendering and user interface technologies, such as Eclipse Sprotty, SVG, and CSS, without the use of abstraction layers. This feature allows for an excellent debugging experience. GLSP also provides a decent feature set, with some implementation needed for adding common editing operations. Nonetheless, GLSP lacks in-depth documentation and community support for troubleshooting.

3.3.1 Testing the Eclipse GLSP

Initially, we found Eclipse GLSP to be a promising choice to begin our project. The framework provided valuable information and an overview of its capabilities. Additionally, there were several example projects available for reference.

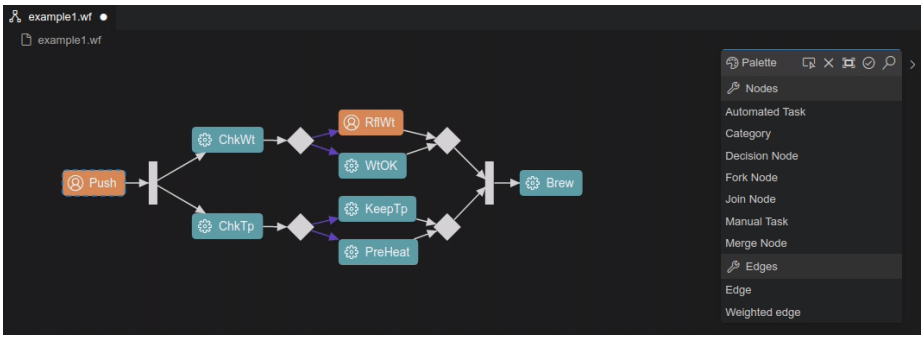


Figure 3.1: Example model from GLSP tutorial

However, when we attempted to modify the example models, we encountered difficulties due to the lack of documentation explaining these examples in detail.

The documentation for Eclipse GLSP proved to be imprecise and unhelpful when it came to assisting developers in testing the framework. Another challenge we faced was the extensive amount of code required to run even a simple example, such as the coffee brewing example shown in Figure 3.1. The size of the codebase for this particular example was relatively large, making it challenging to maintain a potential prototype software in the long run.

This makes the codebase overwhelming and together with poor documentation, we found ourselves scanning the directories of the example without finding any real code to expand and edit to create a CPN model or a CPN editor. We also observed that the codebase for simple examples where relatively large, making a potential software hard to maintain.

3.4 Elm Petri Net editor

Elm is a statically-typed functional language that targets web application development, offering a balance of simplicity and quality tooling, and compiles to JavaScript. It has a syntax similar to Haskell and emphasizes both type safety and developer productivity. Elm was first introduced in 2012 as a thesis by Evan Czaplicki, a former Harvard University student. The language has since then kept evolving and had its last big update in 2019 with version 0.19.1.

Elm boasts of several features that make it stand out as a web development tool. These include its ability to prevent runtime errors in practice through its static typing feature, which validates and corrects errors at compile-time. It also provides user-friendly error messages, supports reliable refactoring, and enforces semantic versioning for all Elm packages automatically. Elm does not inherently provide built-in functionality tailored for modeling purposes. Consequently, significant development time would be required to implement essential editing operations necessary for efficient modeling.

The adoption of Elm for our project did not involve starting from scratch; instead, we built upon an existing codebase available at [24]. The initial development of a Coloured Petri Net (CPN) editor in Elm was initiated by Kent

Inge Fagerland Simonsen around 2017. However, when building on the codebase, encountered several challenges, as significant portions of it had become deprecated. It was discovered through research that Elm underwent continuous updates until 2019, resulting in changes to semantics, syntax and libraries during the period after the project's inception.

Approximately six weeks were dedicated to updating the codebase and resolving various errors. This endeavor exposed a lack of documentation specific to our intended purpose. Despite our efforts, we only managed to create a basic functioning editor with limited capabilities and difficulty in expansion. Consequently, we embarked on the search for an alternative framework to fulfill our listed requirements.

3.5 GoJS

GoJS[26] is a Javascript library for interactive diagrams in modern web browser. It is designed to make it easy to create complex diagrams, with support for a wide range of customization options and predefined functionalities available through its API. With GoJS, developers can create custom node and link types, customize the appearance of individual nodes and links, and add animations to enhance the user experience.

Additionally, GoJS provides support for data-binding, allowing developers to connect graphical objects to model data and automatically update the diagram as the data changes. This feature can help make applications more dynamic and easier to maintain, as changes in the underlying data will automatically be reflected in the diagram. Diagrams in GoJS are conveniently represented in text form using JSON. This approach is highly practical because JSON is a widely adopted and standardized file format for data transfer.

GoJS has a well-sized and active community, which provides great support and resources for developers. The library is continuously updated with new features and improvements, making it more powerful and easier to use. Moreover, GoJS has a highly searchable documentation, which offers clear and concise explanations of the library's features and functionalities, helping developers quickly find the information they need to implement their diagrams. They also have a great set of example models to draw inspiration from and shows the potential of GoJS.

Following a thorough examination of the comprehensive tutorial provided by Northwood Studios on GoJS, showcasing its exceptional editability and wide range of functionalities, it became apparent that this framework possesses the necessary features and capabilities to effectively construct a CPN editor. The tutorial highlights the extensive functionality and flexibility of GoJS, thereby affirming its suitability for fulfilling the requirements of a CPN editor.

3.6 Electron

Electron[10] (formerly known as Atom Shell) is an open source software framework developed and maintained by GitHub. It allows developers to create cross-

platform desktop applications using web-technology such as HTML, CSS and JavaScript. Electron was first released in 2013, and since then, it has gained popularity among developers due to its ease of use and versatility.

The framework allows developers to create desktop applications using the same tools and languages that they use to build web applications. This makes it easy for web developers to enter the desktop development world without needing to learn new programming languages or framework. Electron embeds Chromium and Node.js into its binary, which allows users to maintain one JavaScript codebase and create cross-platform apps that work on Windows, macOS, and Linux.

One of the key features of Electron is its ability to package applications for different platforms, such as Windows, macOS, and Linux, using a single codebase. This means that developers can write an application once and distribute it to multiple platforms without needing to create different versions for each platform.

Electron is a widely adopted framework that powers numerous popular applications, including Visual Studio Code, Slack, and Discord, among others. Its versatility and robustness have made it a favored choice among developers for creating cross-platform desktop applications.

Considering the capabilities of Electron and its successful integration with various tools and libraries, we have identified a potential synergy between Electron and GoJS for our modeling tool development. With Electron as the foundation, we can ensure cross-platform compatibility, enabling users to utilize the modeling tool on different operating systems. GoJS complements this by providing a comprehensive set of features for creating and manipulating visual representation of CPN models.

By leveraging the strengths of Electron and GoJS, we aim to develop a high-quality modeling tool that meets our requirements and provides an intuitive and efficient user experience.

3.7 Candidate software summary

Table 3.1 summarizes the candidate software platforms. It gives an overview of the candidate software platforms, evaluating them on criteria such as modernity, documentation, maintainability, and community support. After careful consideration and analysis of the table, we have determined that GoJS in conjunction with Electron for cross-platform desktop development aligns best with our application requirements.

Framework	**Modern**	**Documentation**	**Maintainability**	**Community**	**Feature set**
Eclipse EMF Core	No	Mediocre	High	Active	Unknown
Eclispe GLSP	Yes	Poor	High	Inactive	Medium
Elm lang	Yes	Poor	Low	Inactive	Small
GoJS	Yes	Good	High	Active	Large

Table 3.1: Comparison of frameworks based on modernity, documentation, maintainability, and community.

GoJS, a modern framework, offers extensive documentation and high maintainability, making it a reliable choice for building our modeling tool. Additionally, it benefits from an active community, which ensures ongoing support and further development. When combined with Electron, we gain the advantage of cross-platform compatibility, enabling users to utilize our modeling tool on various desktop operating systems.

Chapter 4

Design and Implementation

In this chapter, we discuss the design and implementation of the CPN editor, presenting an overview of the modeling environment and introducing innovative solutions for existing features found in CPN Tools.

The CPN editor is designed to provide users with a comprehensive and intuitive modeling environment. It encompasses various components such as the modeling canvas, palette, and tree view, all of which works harmoniously to facilitate the creation and manipulation of CPN models. The modeling canvas serves as the main workspace where users can visually design their models by placing and connecting different elements. The palette offers the main CPN components, enabling users to easily select and add them to their models. The tree view provides a hierarchical representation of the models structure, allowing for convenient navigation and management of components.

Throughout this chapter, we will delve into the details of these design and implementation choices. By introducing a modern and user-friendly approach to CPN modeling, we aim to empower users with a powerful tool that streamlines their modeling processes and enhances their productivity.

4.1 Overview

Figure 4.1 represents a high-level view of our application's architecture and its main components. The ElectronJS desktop application serves as the user interface, which visualizes and runs our modelling tool using GoJS, to create, edit and view CPN Models.

The core functionality of the application revolves around the modeling tool and functionality, which enables the users to design CPN models using intuitive and user-friendly interfaces. Users can define places, transitions, and arcs, specifying the behavior and connections within their models.

To facilitate simulation of the CPN models, the application interfaces with a mock simulator. When the user requests to simulate a model, a message is sent to the simulator, requesting the next simulation step. The simulator then

returns a response containing the state of all nodes, including highlighted transitions and correct markings for places. This feedback provides the user with valuable insights into the behavior and dynamics of the model during simulation.

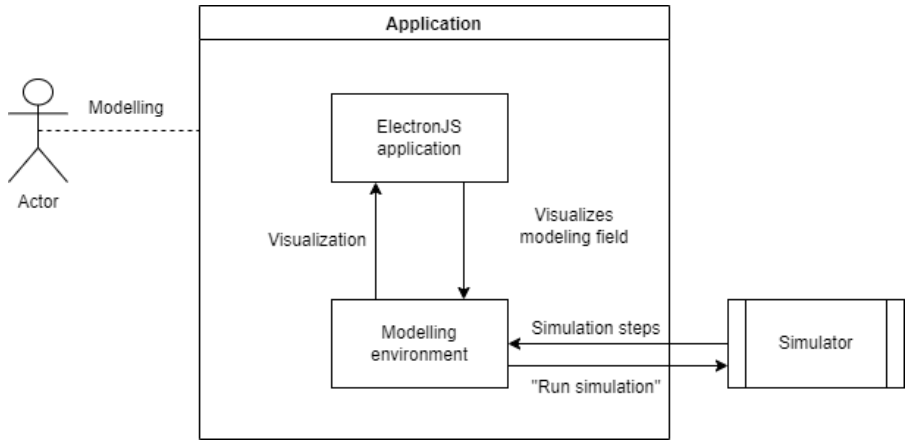


Figure 4.1: High Level view of the application

We will now go through the implementation of features in our application using GoJS. This will show the features that results in creating CPN model of the two-phase commit protocol as show in Figure 4.2.

4.2 Implementation

In the code listings provided, the symbol `$` is utilized following the GoJS coding convention. This practice serves the purpose of reducing code clutter and enhancing code readability. As per this convention, it is customary to commence a file by declaring the constant `const $ = go.GraphObject.make;`. By assigning the `go.GraphObject.make` function to the `$` variable, developers can subsequently employ the `$` symbol in place of `go.GraphObject.make` throughout the file. This approach contributes to the production of cleaner and more concise code, thereby facilitating a more streamlined and manageable coding process within the GoJS framework.

4.2.1 Electron

To begin the development of our CPN editor, we utilized the `npm`(the Node Package Manager)[19] to create a new initial Electron project. Within our Electron project, we created an `index.js` file which serves as the entry point for our application. This file is responsible for starting the Electron application and initializing the main browser window that will display the modeling tool and its associated features.

Shown in Listing 4.1, we define an Electron `BrowserWindow`. When the Electron application is ready it opens a `mainWindow` with a set width and height, and prefences `nodeIntegration` and `enableRemoteModule` set to true and `contextIsolation` set to false.

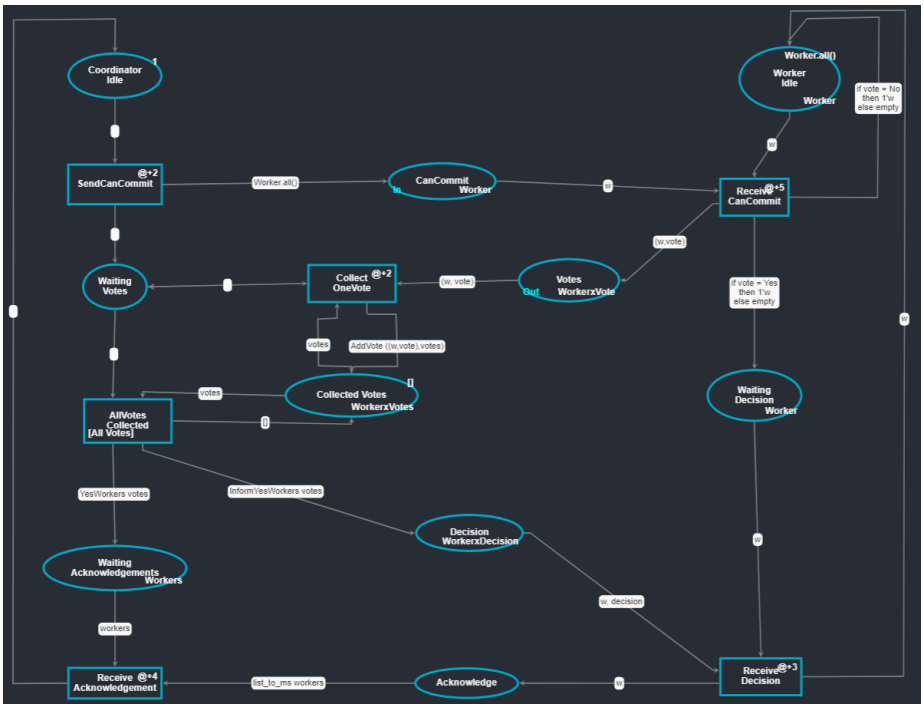


Figure 4.2: Two-phase commit protocol without substitution transitions using GoJS

By setting the `nodeIntegration` property to true, we allow the application to access `Node.js` APIs. This enabled the use of modules and functionalities from `Node.js` within the rendered content.

When the `contextIsolation` property is set to false, it disables the context isolation for the renderer process. Without context isolation, the renderer process has direct access to `Node.js` APIs and can use Electron APIs without the need for a preload script. This allows for more flexibility in terms of accessing system resources and utilizing Electron's full range of capabilities. However, this also increases the risk of potential security vulnerabilities: the renderer process has broader access and potentially execute unsafe code.

Setting the `enableRemoteModule` property to true, it enables the use of the remote module in the application. This allows the application to access Electron's main process modules directly. This can be useful for inter-process communication and accessing Electron specific functionality from within the application.

We define a rule so that closing all Electron browser windows, will make electron quit its execution. When we have done this we have a fully functional application that visualises our application running using HTML, CSS and JavaScript.

```
1  const electron = require("electron");
2  const app = electron.app;
3  const BrowserWindow = electron.BrowserWindow;
4  var mainWindow = null;
5  app.on("ready", () => createWindow());
6
7  function createWindow() {
8    mainWindow = new BrowserWindow({
9      width: 1600,
10     height: 400,
11     webPreferences: {
12       nodeIntegration: true,
13       contextIsolation: false,
14       enableRemoteModule: true,
15     },
16   });
17   mainWindow.loadURL(`file://${__dirname}/index.html`);
18   mainWindow.on("closed", () => {
19     mainWindow = null;
20   });
21 }
```

Listing 4.1: Opening a window using electronjs

The integration of Electron with GoJS proved to be a smooth process, despite being introduced four weeks after initially using GoJS. We successfully incorporated Electron into our project by including an `index.js` file, which serves as the entry point for opening a window that references the `index.html` file. Within the `index.html` file, we seamlessly integrated GoJS and leveraged its functionalities for our modeling tool. The combination of GoJS and Electron worked harmoniously, allow us to harness the interactive and dynamic features of GoJS within the Electron desktop application environment. This integration enabled us to create a seamless user experience creating CPN models.

4.2.2 Object style

The styling of our places and transitions is primarily determined by the `nodeStyle` function. This function is responsible for defining the resizing behavior, as demonstrated in lines 5-7 of Listing 4.2. Initially resizing is enabled, and the size of the resize cells is set to 10x10 pixels. During resizing, the function looks for the object named `"PANEL"` to determine the appropriate behavior for the correct part of the node selected.

In line 9, we bind the `node.location` property to the `loc` property of the node data. This binding is achieved by converting the location using the `Point.parse` static method. This means, if the `Node.location` is changed, it automatically updates the `loc` property of the node data, and vice versa, using the `Point.stringify` static method for conversion.

Additionally, we ensure that the node location is set to the center of each node, providing the most coherent movement for the node. We also link the context menu of the nodes to a predefined context menu, which will be further described in Section 4.2.9. This allows users to access specific styling by right-clicking them.

```
function nodeStyle() {
    return [

      {// Resize the node, with cells to drag
        resizable: true,
        resizeCellSize: new go.Size(10, 10),
        resizeObjectName: "PANEL",
      },
        new go.Binding("location", "loc", go.Point.parse).
            makeTwoWay(
        go.Point.stringify
      ),
      { locationSpot: go.Spot.Center },
      { contextMenu: context.diagramContextMenu() },
    ];
  }
```

Listing 4.2: Styling nodes

4.2.3 Places

The places in our application have two marking fields: The initial marking and the current marking. The initial marking field is used to set the initial marking of the place, used to indicate the initial state of the CPN model. On the other hand, the current marking field displays the current marking of the place. This field remains hidden until the node receives a marking from the mock simulator. When the marking is received, this field becomes visible, displaying the current marking of the place.

In addition to the marking fields, there is a colour set inscription field with placeholder `UNIT`. This field specifies the type of association with the specific place. This will be used in simulation to to determine the type of token that can be put in the place.

The places also have an I/O field, which is in use if the place is a port or socket for a substitution transition. This field will be used to manage input/output communication between different part of the CPN model.

These fields and properties on the places contribute to the overall functionality and behavior of the CPN editor, providing the user with the ability to define initial states, track current marking, specify unit types, and manage input/output operation in their models.

In Figure 4.3, we can see two places: the `Coordinator Idle` place and the `Waiting Votes` place. The `Coordinator Idle` place is represented with an initial marking of 1, indicating its initial place. `Waiting Votes` place has its colour set set to `Workers`.

Both places exhibit the default style for places in our application. The default style includes the visual representation of places as circles and by default a shade of cyan.

Listing 4.3 showcases the initial implementation of places in our CPN editor, focusing on their style and shape. This portion of the code defines the visual

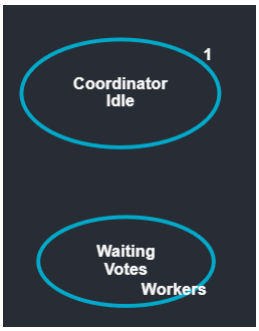


Figure 4.3: Example of Two places in our solution

appearance of places, such as their size, colour, and shape. It sets the foundation for the further enhancements and functionality added.

It is important to note that the code snippet in Listing 4.3 specifically addresses the visual representation of places, and it does not encompass the implementation of inscription fields. The implementation of inscription fields, which are used to display and manipulate information associated with places and transitions, will discussed in Section 4.2.5.

The implementation of the place template begins by creating an object named `Place`. This object contains a node, in the style of a `Table`. Within the node, we add a `Panel` and a `Shape` to the `Panel`. The Panel becomes visible when the place is selected, as the selection border highlights the panel. The `Shape` defines the shape and colors of the node, with the node color matching the modeling canvas for visually cohesive appearance.

In the code snippet, you can see that we bind the properties of the `go.Shape` element, such as `figure`, `stroke`, and `fill`. These bindings allow us to change these values dynamically, which can be useful for future features or interactions within the CPN editor. For example, the bindings could be used to modify the figure shape, change the stroke colour, or update the fill color based on certain conditions or user actions.

```
myDiagram.nodeTemplateMap.add(
    "Place",
    $(
      go.Node,
      "Table",
      nodeStyle(),
      $(
        go.Panel,
        "Auto",
        { name: "PANEL" },
        $(
          go.Shape,
          "Ellipse",
          {
            name: "NODESHAPE",
            fill: "#282c34",
            stroke: "#00A9C9",
            strokeWidth: 3.5,
          },
```

```
20              new go.Binding("figure", "figure"),
21              new go.Binding("stroke", "color"),
22              new go.Binding("fill")
23            ),
24            ...
```

Listing 4.3: Start of place implementation

By setting up these bindings, we create a flexible and customizable template for places in CPN editor. It provides the foundation for incorporating additional functionality and allows for easy modification of the place's visual attributes as needed.

4.2.4 Transitions

The transitions in our CPN editor have several inscription fields: guard, time, code, and priority. These are the same as mentioned earlier in 2.1

The guard field is used for specifying guard expressions. Guard expressions define conditions that must be satisfied for a transition to be enabled and fired. They allow for conditional behavior in the CPN model.

The time field is used to simulate the duration of a specific task. It allows us to add to the total time of the simulation. The syntax for the time inscription is `@+x`, where `x` represents the time specified in integers. This helps in modeling time-dependent processes and capturing the time aspect of the CPN model.

The code segments in the transition represents executable code that is executed when the parent transition occurs. These code segments can include input and output variables, and perform specific actions or calculations. They enable execution of custom logic or algorithms associated with the transition.

Lastly, the priority field is used to specify the priority of a transition. It is often set with predefined values such as `P_HIGH`, `P_NORMAL` or `P_LOW`. By default, these variables are assigned the respective values of 100, 1000, and 10 000. The priority of a transition determines its precedence in cases where multiple transitions are enabled simultaneously.

Figure 4.4 illustrates the representation of the `Send CanCommit` transition in our CPN editor. In this example, the transition is depicted without any inscription between `Coordinator Idle` place(with an initial marking), the `CanCommit Message` place, and the `Waiting Votes` place. It is worth to note that in the absence of a simulator, the transition is not currently highlighted as it would be in a fully functional model.

Although the highlighting is not present in this specific example due to the lack of a simulator, the design and implementation of the transition in our CPN editor follows the established conventions and principles of CPN modeling.

These inscription fields provide additional flexibility and functionality to the transitions in our CPN editor, allowing for more sophisticated modeling and simulation of processes. We discuss the implementation of inscription fields in Section 4.2.5.

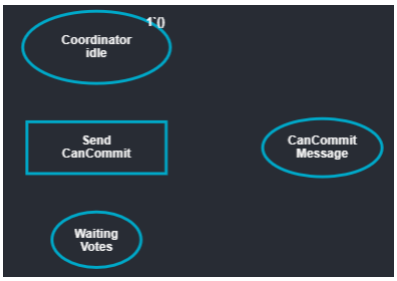


Figure 4.4: Example of the transition Send CanCommit

4.2.5 Shared attributes in nodes

Both places and transitions in our CPN editor have a value called `category`, which serves to indicate their respective node types. The category value is automatically assigned as either `Place` or `Transition` and cannot be edited by the user. This categorization is also used when connecting arcs between nodes. If the user attempts to connect two nodes of the same category, no port indicator will be displayed, and the connection between them cannot be established

In addition to the category, both places and transitions have their own editable name field. This field allows users to provide descriptive names for the nodes, enabling them to provide information about the specific functionality or purpose of each node in the model. The name serves a way to enhance the understandability and clarity of the CPN model, making it easier for users to identify and comprehend the actions or meaning associated with each node.

In the provided Listing 4.4, we demonstrate an example of an inscription field implementation that can be for both transitions or places. This specific code snippet showcases the integration of the guard expressions for transitions.

To begin, we create a new TextBlock object, which is responsible for displaying the guard expression text. By default this TextBlock is styled according to the predefined TextStyle. We position the TextBlock in the top left corner of the node and align the text to the left within the TextBlock.

The `wrap` property is utilized to control the wrapping behaviour of the node. We assign `wrap` with the value `go.TextBlock.WrapFit`. By doing this the width of the node will adjust to accommodate the longest line of text within the TextBlock, ensuring proper visibility of the guard expression.

We establish a binding mechanism that allows dynamic updates to the guard expression. This enables the ability to modify the guard text through a textarea input field or potentially through alternative methods used in the future, providing flexibility and ease of use for users interacting with the CPN editor.

```
$(
  go.TextBlock,
  textStyle(),
  {
    alignment: go.Spot.TopLeft,
    textAlign: "left",
    margin: 8,
    wrap: go.TextBlock.WrapFit,
    editable: true,
  },
  new go.Binding("text", "guard").makeTwoWay()
),
```

Listing 4.4: Implementation of guard expression field

The templates for places and transitions in our implementation share many similarities, but they are distinguished by their naming, shape, and their specific set of inscriptions they contain.

Both templates incorporate multiple TextBlocks, similar to the example provided in Listing 4.4, which demonstrates the implementation of a guard expression on transitions. These TextBlocks are responsible for displaying various inscriptions associated with the places and transitions, such as names, guard expressions, time inscriptions, and code segments.

Figure 4.5 provides a visual representation of a selected transition and a place, showcasing the default inscription included in our CPN editor. These inscriptions provide users with essential information about the nodes and their associated behaviors, enhancing the usability and comprehensibility of the modelling environment.

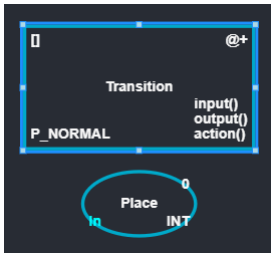


Figure 4.5: Example of place and transition with their inscription fields

4.2.6 Ports

Mapping transitions and places in our implementation involves assigning ports to these nodes. Ports serve as connection points for arcs, allowing arcs to be drawn to and from transitions and places. Each transition and place in our CPN editor has four ports, positioned on all sides of the nodes: top, bottom, left and right.

To provide visual feedback to users, we have implemented a hover effect on the ports of the nodes. When hovering over a port, a purple shape is highlighted as demonstrated in Figure 4.6, indicating that an arc can be dragged from that

spot. While dragging an arc to another node, a new purple shape appears, indicating a valid location to place the arc.

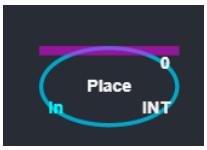


Figure 4.6: Example of visible port

Listing 4.5 demonstrates an example implementation of ports on places and transitions. The `makePort` function is used for creating ports and accepts several parameters. The `name` parameter is used to identify the port, while `align` determines the side to which the port aligns. The `spot` parameter specifies the position of the port. Lastly, the boolean parameters `output` and `input` define whether the port is meant for output arcs, input arcs, both, or neither.

This port implementation enables users to establish connections between transitions and places, allowing for the creation of complex and interconnected CPN models within our modelling tool.

```
1 // implementation of ports on nodes and transitions.
2 makePort("T", go.Spot.Top, go.Spot.Top, true, true),
3 makePort("L", go.Spot.Left, go.Spot.Left, true, true),
4 makePort("R", go.Spot.Right, go.Spot.Right, true, true),
5 makePort("B", go.Spot.Bottom, go.Spot.Bottom, true, true),
```

Listing 4.5: Implementation of ports

4.2.7 Arcs

In our CPN editor, the implementation of arcs is an essential part of creating CPN models. Arcs serve as connections between places and transition, indicating the flow of tokens and information.

The arc template we have developed is designed to accommodate different types of arcs. By default, arcs are represented in a gray color. When hovering over an arc, it highlights to indicate its selection.

We have enabled the feature making users able to reshape and resegment arcs. Users can adjust the shape and segments of arcs needed to create the desired visual representation of their CPN models. Additionally, arcs can be relinked to different places or transitions if required, providing flexibility in modeling connections.

To handle the different types of arcs, we have implemented a single arc template that includes the most common arrowhead options, such as default, inhibitor, reset, and bidirectional arcs. By default, the arc template uses a standard arrowhead pointing towards the place or transition it connects into. By using the context menu, users can change the type of arc, which updates the visibility of arrowheads accordingly. For example, selecting an inhibitor arc would make the inhibitor arrowhead visible and the previous arrowhead transparent. The same principle applies to bidirectional arcs, which are represented by a transparent

arrowhead near the place or transition from which the arc originates, but can be made visible when needed.

There are different kinds of arcs. We have solved the implementation of this by just having one arc template. This arc template has all the kinds of arrowheads on it; default, inhibitor and reset arc, but also the option of bidirectional arc. By default we have a standard arrowhead towards the place or transition the arc is pointing at. On changing the type of arc using the context menu, we make the arrowhead we need visible, and the previous arrowhead transparent. The same goes for bidirectional arc, which is just an, by default transparent, arrowhead by the place or transition the arc originates from that we could make visible when needed.

Arcs has a ruleset that checks their connecting nodes. We make sure that if the node the arc originates from is a place, then the connecting node must be a transition. This also goes for the other way around. So if the originating node is a transition, the node that the arc is connected to, must be a place.

Figure 4.7 illustrates an example of an arc in our CPN editor, connecting the `Send CanCommit` transition with the `CanCommit` place. The arc represents the flow of tokens and information between these two elements in the CPN model.

This arc demonstrates the essential role of arcs in our CPN editor, enabling the modeling of relationships and interactions between transitions and places. It helps to define the behavior and dynamics of the CPN model, capturing the flow of tokens or data as it moves through the system.

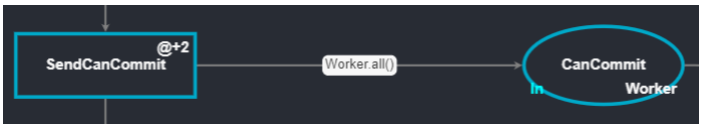


Figure 4.7: Arc in GoJS application between Send CanCommit and CanCommit

Listing 4.6 provides demonstration of parts of the link template in our CPN editor. The link template defines the behavior and appearance of arcs connecting places and transitions.

In lines 4-17, we define the initial object properties of the arc. These properties govern various aspects such as avoiding places and transitions using routing `go.Link.AvoidsNodes`, rounding corners, and setting the length of the end segment using `toShortLength`. By avoiding places, transitions, and comments, the arcs provide a visually pleasing and clear representation of the model.

The link template allows for interactive features such as relinking, reshaping, and resegmenting. Users can modify the structure and layout of arcs as needed, providing flexibility in designing their CPN models. The context menu is referenced to handle right-click events, enabling substituting of arc types.

The code also includes event handles for mouse enter and mouse leave events, which control the highlighting of the arcs when the cursors hover over a specific one. The highlight shape, represented by a blue colour with an alpha blending value of 0.2, provides visual feedback to the user without disrupting the entire shape of the arcs. The highlight is a `go.Shape` with a slightly larger

`strokeWidth` value than the default arc shape, which is 1. The highlight shape is initially set to be transparent.

```
myDiagram.linkTemplate = $(
  go.Link, //the whole link panel
  {
    routing: go.Link.AvoidsNodes,
    curve: go.Link.JumpOver,
    corner: 5,
    toShortLength: 4,
    relinkableFrom: true,
    relinkableTo: true,
    reshapable: true,
    resegmentable: true,
    contextMenu: context.linkContextMenu(),
    // mouse-overs subtly highligh links:
    mouseEnter: (e, link) =>
      (link.findObject("HIGHLIGHT").stroke = "rgba(30,144,255,0.2)"
          ),
    mouseLeave: (e, link) =>
      (link.findObject("HIGHLIGHT").stroke = "transparent"),
  },
  new go.Binding("points").makeTwoWay(),
  $(
    go.Shape, //the highligh shape, normally transparent
    {
      isPanelMain: true,
      strokeWidth: 8,
      stroke: "transparent",
      name: "HIGHLIGHT",
    }
  ),
  ...
```

Listing 4.6: Template for arcs

This flexible arc template allows users to create different types of arcs within their CPN models and provides intuitive visual cues for understanding the flow and nature of each arc.

We further discuss the implementation of changing arrowheads using the context menu under Section 4.2.9

4.2.8 Arc inscriptions

Arc inscriptions provide additional conditions associated with arcs in CPN models. In our implementation, we position the arc inscription in the middle of the arc to make sure longer inscription does not overlap the places or transitions being connected by the arc.

As shown in Figure 4.8, the arc inscription specifies that if the incoming token from the `Receive CanCommit` place is of the type `Vote`, the transition will send a token `1'w`. Otherwise, if there is no incoming token or the incoming token is not a `Vote`, the transition will have an empty output.

In the implementation of arc inscriptions seen in Listing 4.7, we use a panel named `LABEL` to contain the inscription text. The `segmentFraction` property determines the position of the arc inscription along the arc and is a floating-point

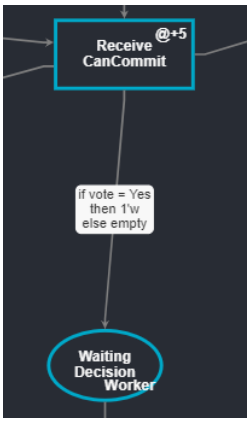


Figure 4.8: Example arc inscription in GoJS solution

number between 0 and 1, where 0 represents the starting point of the arc and 1 represents the ending point of the arc. By setting the `segmentFraction` to 0.5, we instruct the arc inscription to position itself in the middle of the arc, ensuring that it is visually centered between the connected places and transitions. This allows us to maintain consistent positioning of the arc inscription, regardless of the length or shape of the arc. It ensures that the inscription remains easily readable and visually aligned with the flow of the arc.

In the implementation of the arc inscription, we have set up a binding for the visibility property. This binding allows us to control the visibility of the arc inscription based on its content. We still render inscription field, as it would be hard for the user to guess the location of where they should click to start editing. An empty inscription field is rendered as a simple white space in the middle of the arc. This is demonstrated in Figure 4.11 in the next Section.

Last part of the implementation of the arc inscription field, we use a TextBlock element to display the text. The name `expr` is used as both the placeholder text and as the identifier for the TextBlock element. This choice of name aligns with the default placeholder for arc inscriptions used in CPN Tools. We have customized the TextBlock element by setting text alignment to be centered, set the default font to be black 10 pt helvetica. If helvetica is not available in the runtime environment or the system running for some reason, it will fall back on arial. Similarly, if arial is not available, it will fall back on sans serif. Lastly, we enable the editing, so that users can modify the inscription.

```
$(
  go.Panel,
  "Auto",
  {
    visible: true,
    name: "LABEL",
    segmentFraction: 0.5,
  },
  new go.Binding("visible", "visible").makeTwoWay(),
  $(
    go.Shape,
    "RoundedRectangle", //label shape
    { fill: "#F8F8F8", strokeWidth: 0 }
  ),
  $(
    go.TextBlock, // the label
    "expr",
    {
      textAlign: "center",
      font: "10pt helvetica, arial, sans-serif",
      stroke: "#333333",
      editable: true,
    },
    new go.Binding("text").makeTwoWay()
  )
)
```

Listing 4.7: Implementation of arc inscription

By including arc inscriptions, we can capture and represent various conditions, or expressions associated with the flow of tokens between places and transitions in our CPN editor. This provides a richer and more expressive modeling capability, allowing users to define complex behaviour and rules within their CPN models.

4.2.9 Context menu

The context menu in our implementation serves the purpose of allowing users to edit the visualization of places, transitions, and arcs. It can be accessed by right-clicking on any object on the modeling canvas.

The context menus consist of buttons with two different functionalities: `ColorButtons` and `ArrowButton` functions. These functions are responsible for creating buttons specific to each context menu. However, there is a common function that handles the changes requested by the user and commits them to the model. This function is the `ClickFunction` and is shown in Listing 4.8.

When a button in a context menu is clicked, the `ClickFunction` is invoked with the relevant input parameters: `propname` represents the property to be changed, and `value` represents the new value to which the property is going to be changed. To prevent unintended changes, we set the `e.handled` property to true, indicating that the event has been handled and further changes should be stopped.

After ensuring that only the intended changes are made, we send a commit to the model. The set function is used to locate the object that was clicked on and

find the specific property that needs to be changed. Once found, the property is updated with the new value provided.

```
function ClickFunction(propname, value) {
  return (e, obj) => {
    e.handled = true;
    e.diagram.model.commit((m) => {
      m.set(obj.part.adornedPart.data, propname, value);
    });
  };
}
```

Listing 4.8: Function handling a selection in one of the context menus

To summarize, the context menu functionality allows users to customize the visual aspects of the elements in the modeling canvas. The associated functions, such as `ColorButtons` and `ArrowButton`, generate the buttons in the context menu, while the common `ClickFunction` ensures that the desired changes are properly committed to the model.

Context menu for places and transitions

For places and transitions, we can use the context menu to change the color of the border, or fill the inside of our node/transition of choice. We also have implemented a button that returns the color scheme of the selected place/transition to default. The context menu is shown in Figure 4.9.

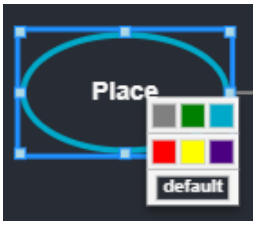


Figure 4.9: Example of open context menu on a place

Listing 4.9 demonstrates the template function for creating buttons with the different colours in the context menu. The function takes the two inputs: `color`, which represents the colour we want to change the property to, and `propname`, which denotes the property we intend to modify.

The function first checks if a `propname` is provided. If it is not, the function sets the `propname` to `color`. This simplifies the implementation by automatically associating the `propname` to color.

Next the function checks if the provided color is equal to `cyan`. If it is, the function adjust the color to the correct shade of cyan we have used by default. This adjustment is made to ensure better code readability and maintain consistency with the desired color representation.

The `ColorButton` function provides a template for creating buttons with different colours in the context menu for places and transitions. The function allows for the customization of properties by associating the chosen colour with the target property, simplifying the implementation process.

After the initial checks, the function in Listing 4.9 creates the shape of the button. Starting with the function setting the size of the buttons. The function then handles the `mouseEnter` and `mouseLeave` events to add a subtle highlight effect to the buttons when the mouse cursors hovers over them. This visual feedback enhances the user experience by indicating interactivity.

When a button is clicked, either by left-clicking or right-clicking, the function sends the corresponding values of the button and object of the context menu, property name and the chosen colour, to the `ClickFunction` mentioned in Listing 4.8. This ensures that the desired changes are properly processed and committed to the model. The shape containing this functionality is then returned to the rest of the context menu.

```
function ColorButton(color, propname) {
  if (!propname) propname = "color";
  if (color === "cyan") color = "#00A9C9";
  return $(go.Shape, {
    width: 16,
    height: 16,
    stroke: "lightgrey",
    fill: color,
    margin: 1,
    background: "transparent",
    mouseEnter: (e, shape) => (shape.stroke = "dodgerblue"),
    mouseLeave: (e, shape) => (shape.stroke = "lightgrey"),
    click: ClickFunction(propname, color),
    contextClick: ClickFunction(propname, color),
  });
}
```

Listing 4.9: Template for colour changing buttons in context menus

Listing 4.10 shows the function responsible for changing the filling and border colors of the selected place or transition back to the default values. This function is associated with a button in the context menu, which has the same background color as the objects and displays the text ”default”.

The functionality of this function is similar to the previously mentioned click function. However, instead of sending individual property changes to the model, this function combines two set commands into a single commit. The set commands are used to reset the filling and border colors of the selected object to their default values. Adding both these changes in a single commit ensures that the default color restoration is performed simultaneously, avoiding any delay or inconsistency in the visual appearance of the objects.

```
function defaultClickFunction() {
  return (e, obj) => {
    e.handled = true;
    e.diagram.model.commit((m) => {
      m.set(obj.part.adornedPart.data, "color", "#00A9C9");
      m.set(obj.part.adornedPart.data, "fill", "#282c34");
    });
  };
}
```

Listing 4.10: Function for changing back to default colour

For transitions, we have added an extra button for linking a transition to the corresponding submodule. This is related to the implementation of substitution transitions. The extra button in the context menu executes the `linkSubModule` function. The implementation of the `linkSubModule` function and related functionality to substitution transitions can be seen in Section 4.2.11. The context menu for transition is displayed in Figure 4.10

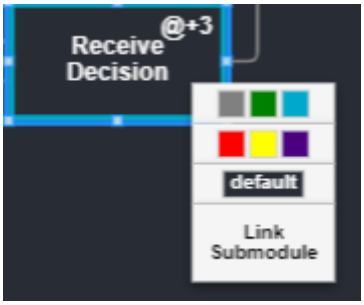


Figure 4.10: Context menu of a transition

Context menu for arcs

Context menu for arcs are used to change the arc type. In practice we change the arrowhead of an existing arc, as mentioned in Section 4.2.7. This is done by making all other arrowheads transparent and the arrowhead we want visible.

Figure 4.11: Example of a open context menu on an arc

Implementing the arrowheads of the arcs, SVG paths are used to define their shapes. This choice of format allows for easy scalability and the ability to modify their colors. To simplify the code and make it more manageable, a numbering system is introduced to identify the different arrowhead shapes.

In the code, each arrowhead shape is assigned a corresponding geometry string, denoted as `geo`. These geometry strings represents the SVG path data that defines the shape of the arrowhead. By associating a number with each arrowhead, the code becomes more concise and easier to maintain.

When a user clicks on an arrowhead button in the context menu, the corresponding number is sent as the value, and `"dir"`is sent as the `propname` to the click function. This allows the click function to identify which arrowhead shape is selected and perform necessary changes to the model.

By organizing the arrowhead shapes in this particular way and utilizing the numbering system, the code becomes more modular and flexible. It allows for easy addition or modification of arrowhead shapes in the future without the need for extensive code changes.

```
function ArrowButton(num) {
  // Single arrow
  var geo = "M0 0 M16 16 M0 8 L16 8  M12 11 L16 8 L12 5";
  if (num === 0) {
    // No arrow
    geo = "M0 0 M16 16 M0 8 L16 8";
  } else if (num === 2) {
    // double arrow
    geo = "M0 0 M16 16 M0 8 L16 8  M12 11 L16 8 L12 5  M4 11 L0 8
        L4 5";
  } else if (num === 3) {
    // reset arc
    geo = "F M 0 0 M 16 16 M 8 8 M 12 11 L 16 8 L 12 5 M 4 11 L 8 8
         L 4 5";
  } else if (num === 4) {
    // Inhibitor arc
    geo = "F M 0 0 M 16 16 M 4 8 m 0 0 a 2 2 0 1 0 7 0 a 2 2 0 1 0
        -7 0 ";
  }
  return $(go.Shape, {
    geometryString: geo,
    margin: 2,
    background: "transparent",
    mouseEnter: (e, shape) => (shape.background = "dodgerblue"),
    mouseLeave: (e, shape) => (shape.background = "transparent"),
    click: ClickFunction("dir", num),
    contextClick: ClickFunction("dir", num),
  });
}
```

Listing 4.11: Function for changing arrowheads on arcs

Listing 4.12 is showing the implementation of the arrowhead themselves and not the context menu selection of the arrowhead. This listing we see a new arrowhead geometry definition, specifically for the reset arc. This geometry represents the shape of the arrowhead and is created using a series of path commands in SVG format.

On lines 6-15, a shape element is created within the `linkTemplate`. This shape uses the arrowhead geometry `Reset` and is initially set to be invisible. The color of the arrowhead is defined as grey.

The visibility of the arrowhead shape is bound to the value of the `dir` property. By using this binding, the visibility of the shape can be dynamically controlled based on the value of the `dir` property. When the `dir` property is set to 3, which is the number assigned to the reset arc in the context menu selection, the visibility of the arrowhead shape will be set to true according to the defined predicate.

This implementation allows for the dynamic display of different arrowhead shapes based on the value of the `dir` property. By changing the property through the context menu, the corresponding arrowhead shape will become visible, providing visual feedback to the user for the type of arc.

```
go.Shape.defineArrowheadGeometry(
  "Reset",
  "F M 19 6 M 12 12 L 19 6 L 12 0 L 15 6 L 12 12 M 3 12 L 10 6 L 3
      0 L 6 6 L 3 12"
);

$(
  go.Shape,
  {
    toArrow: "Reset",
    visible: false,
    fill: "gray",
    stroke: "gray",
  },
  new go.Binding("visible", "dir", (dir) => dir === 3)
)
```

Listing 4.12: Custom arrowhead and selecting it

4.2.10 Tree view

The tree view within our application serves as a visual representation of the underlying file hierarchy of the model the user is modeling. As depicted in Figure 4.12, the tree view displays the structure of the model in a hierarchical manner, starting with the outermost and most abstracted model file, which in the case of our example is the `twophase.json` file.

Within the `twophase.json` file, we can observe two submodules: The Coordinator and the Workers. These submodules are presented as nested elements within the tree view, reflecting the hierarchical relationship between them and the main model. This hierarchical representation allows users to easily navigate and understand the models components.

The tree view itself is implemented using GoJS, which provides the necessary functionality for rendering the hierarchical structure and interactions. Due to the security measure of "Same Origin Policy"[5], retrieving the directory structure requires user interaction. The Same Origin Policy restricts JavaScript from making requests to resources that are not from the same origin as the web page itself. This policy is implemented to prevent potential security threats, such as cross-site scripting attacks and unauthorized access to local files.

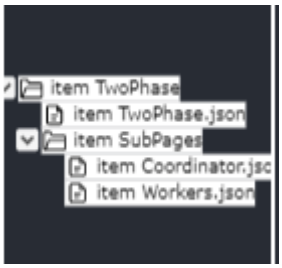


Figure 4.12: The tree view of the two-phase commit protocol

The tree view in our application functions as an independent diagram, and therefore, it requires specific properties to be set in order to customize its ap-

pearance and behavior. In Listing 4.13, we define these properties to ensure optimal visualization and interaction within the tree view.

Firstly, we disable any editing capabilities for the tree view model since its purpose is solely to provide links to different models and offer an overview of the current model the user is working on.

To control the layout of the tree view, we utilize various properties. The `alignment` property determines the alignment of parent nodes to their children. In our case, we set it to `AlignmentStart`, which places the parent node above its children, close to the first child.

The `angle` property defines the direction in which the tree grows. By setting `angle` to 0, we ensure that the tree grows downwards, providing a top-to-bottom layout.

To control the spacing and positioning of nodes within the tree view, we set the `compaction` property to `CompactionNone`, which ensures a consistent spacing style without compacting subtrees.

The `layerSpacing` property determines the distance between layers, specifically the parent node and its child nodes. We set `layerSpacingParentOverlap` to 1, allowing the nodes to be placed closely together.

The `nodeIndentPastParent` property specifies the additional space given to the first child node when positioned relative to its parent.

The `nodeSpacing` property defines the space between nodes within a layer, ensuring an appropriate visual separation.

Lastly, since ports in the tree view is not necessary, we disable the automatic adjustment of port spots based on the angle by setting `setsPortSpot` and `setsChildPortSpot` to false.

```
myTreeView = $(go.Diagram, "myTreeDiv", {
  allowMove: false,
  allowCopy: false,
  allowDelete: false,
  allowHorizontalScroll: false,
  layout: $(go.TreeLayout, {
    alignment: go.TreeLayout.AlignmentStart,
    angle: 0,
    compaction: go.TreeLayout.CompactionNone,
    layerSpacing: 16,
    layerSpacingParentOverlap: 1,
    nodeIndentPastParent: 1.0,
    nodeSpacing: 0,
    setsPortSpot: false,
    setsChildPortSpot: false,
  }),
});
```

Listing 4.13: Treeview properties

In the projects tree view, there are features that allow users to collapse and expand the tree nodes. These functionalities enhance the users ability to navigate and organize the tree structure.

In Listing 4.14, we can see the implementation of these features. The code snippet demonstrates an event handler for the double-click event on a tree node.

To ensure the safety of the operation, the event handler first checks if the selected node has children that can be expanded or collapsed, this prevents any unexpected behavior when attempting to perform these actions on nodes in the tree view without children.

Next, the code checks the current state of the nodes children. If the children are already expanded, executing the double-click action will collapse them, making the child nodes appear under the selected node. Conversely, if the children are currently collapsed, double-clicking on the node will expand them, displaying the child nodes beneath the selected node.

Lines 16-23 in the provided code listing shows the customization applied to the appearance of the expander buttons in the tree view. The expander buttons are predefined buttons specifically designed for tree models.

To customize the expander buttons, the code modifies the figure displayed on the button based on the state of the tree node. When the tree node is expanded, the figure of the button is changed to a downwards-facing arrow, indicating that the child nodes are currently visible. On the other hand, when the tree node is not collapsed, the figure of the button is changed to an arrow facing right, indicating that the child nodes are hidden. Furthermore, the code sets the code color of the expander button to `whitesmoke` and removes the stroke, or the border, around the button.

For more visual information to the user, the tree view incorporates different figures to represent the various types of tree nodes, such as directories or files, in a more intuitive manner. This is achieved through the binding mechanism that connects the current state of each tree node to the `imageConverter` function.

The `imageConverter` function determines the appropriate figure to display for each node. It examines the type of node, distinguished between leaf nodes, for files, and branch nodes, for directories. Based on this evaluation, the `imageConverter` function dynamically assigns the corresponding figure that accurately represents the status of the node.

By incorporating collapsing and expanding functionalities, the tree view enhances the usability and navigation experience for users. The visual representation of different nodes, such as directories and files, through distinctive icons enables users to quickly differentiate between them. For instance, an opened directory is depicted with an icon resembling a collapsed folder, while file nodes have their own distinct icon.

This approach empowers users to effectively manage the visibility and organization of the tree structure according to their specific requirements. They can expand or collapse nodes as needed, allowing for a more streamlined and personalized view of the project components. As a result, the tree view becomes a user-friendly tool that facilitates seamless navigation through the project.

```
doubleClick: (e, node) => {
  var cmd = myDiagram.commandHandler;
  if (node.isTreeExpanded) {
    if (!cmd.canCollapseTree(node)) return;
  } else {
    if (!cmd.canExpandTree(node)) return;
  }
  e.handled = true;
  if (node.isTreeExpanded) {
    cmd.collapseTree(node);
  } else {
    cmd.expandTree(node);
  }
},
...
$("TreeExpanderButton", {
// customize the button's appearance
  _treeExpandedFigure: "LineDown",
  _treeCollapsedFigure: "LineRight",
  "ButtonBorder.fill": "whitesmoke",
  "ButtonBorder.stroke": null,
  _buttonFillOver: "rgba(0,128,255,0.25)",
  _buttonStrokeOver: null,
}),
...
go.Picture,
  {
    width: 18,
    height: 18,
    margin: new go.Margin(0, 4, 0, 0),
    imageStretch: go.GraphObject.Uniform,
  },
  // bind the picture source on two properties of the Node
  // to display open folder, closed folder, or document
  new go.Binding("source", "isTreeExpanded", imageConverter).
      ofObject(),
  new go.Binding("source", "isTreeLeaf", imageConverter).ofObject()
```

Listing 4.14: Functionality handling collapsing tree view

We provide the ability to link to substitution transitions using the tree view. This functionality is implemented using the event handler and the `fetchFile` function, shown in Listing `code:fetchfile`.

The `contextClick` event handler is defined within the `nodeTemplate` for the treemodel. When the user right-clicks on a file node in the tree view, this event handler is triggered. It then passes the path of the clicked file to the `fetchFile` function. The `fetchFile` function utilizes the Fetch API in JavaScript to retrieve the contents of the file specified by the path. Once the file is successfully retrieved, the function extracts the data and updates the model to represent the model corresponding to the file.

This feature allows for easy navigation through the tree view, selecting specific files, and seamlessly switch to associated models. It enhances the flexibility, modularity, and interactivity of the application, allowing users to explore different parts of their model in a convenient manner.

```
contextClick: (e, node) => {
  if (node.isTreeLeaf) {
    fetchFile(node.data.path);
  }
},

function fetchFile(path) {
  fetch("../" + path)
    .then((res) => res.json())
    .then((data) => (myDiagram.model = go.Model.fromJson(data)));
}
```

Listing 4.15: Open submodule from file

To overcome the restrictions imposed by the "Same Origin Policy" and enable the loading of file trees in our application. The solution is depicted in Listing `code:loadTree` along with defining the tree structure.

The solution involves the usage of a select file button located below the JSON representation of the model. When the user clicks this button and selects a folder, a method is triggered to create a file tree based on the files within the selected directory. This method receives the folder and subfolder names from the file path.

The file tree structure is then stored as an object and assigned to the `window.tree` property. This allows us to access and reference the file tree structure later in the application. Once the user has selected a folder and the file tree structure is available in the `window.tree` property, they can utilize the show folder button. Clicking this button triggers the loading of the file tree from `window.tree` into the file tree canvas, where it can be displayed and interacted with.

In the implementation of creating the file tree described above, the file is represented by the `fileObj` object. This object serves as the container for storing information about the separate files. The filename itself serves as a key to uniquely identify the file within the file tree structure.

Additionally, the parent folder of the file is also stored as a property within the `fileObj` object. This is used to establish the hierarchical relationship between files and folders within the file tree.

The files path is also preserved as a property in the `fileObj` object. This path information can be used to locate and access the file when utilizing the file tree structure to read or perform other operations on the file.

```
function readDirectory(event) {
  const files = event.target.files;
  const directory = files[0].webkitRelativePath;
  const fileTree = createFileTree(files);

  console.log("Selected directory:", directory);
  console.log("File tree:", fileTree);
  console.log(JSON.stringify(fileTree, null, 2));
  window.tree = fileTree;
}

function loadTree() {
```

```
13    myTreeView.model = new go.TreeModel(window.tree);
14  }
15
16  const fileObj = {
17    key: fileName,
18    parent: currentDir.key,
19    path: filePath,
20  };
```

Listing 4.16: Functionality for loading the tree

4.2.11 Substitution transitions

The concept of substitution transitions has gone through some changes to enhance modularity in our version of the CPN editor compared to CPN Tools. In our editor, a substitution transition is designed to serve as a transition element, while the underlying model associated with it is stored in a separate file. This approach allows for better organization and separation of concerns within the overall system.

As described in the previous Section 4.2.10, we have implemented the functionality for the user to open the corresponding models using the tree view. This provides a convenient way to access and explore the underlying models associated with each substitution transition.

In addition, we have also implemented the possibility of adding a connection between transitions and a corresponding submodule. This is done using the context menu option, `Link Submodule`. Listing 4.17 exhibits the connection implementation of this feature.

The process of linking a transition is implemented using the `dialog.showOpenDialog` function in Electron. Electron opens the file structure on the users computer. Here the user can select the corresponding submodule JSON file to the transition. The function will return the file path and add it to the data of the selected transition. The user can open the related submodule by double clicking on a transition linking to a submodule using the fetch method.

```
1   dialog
2     .showOpenDialog({
3       title: "Select File",
4       properties: ["openFile"],
5       filters: [{ name: "JSON", extensions: ["json"] }],
6     })
7     .then((result) => {
8       if (!result.canceled && result.filePaths.length > 0) {
9         var fileURL = "file:///" + result.filePaths[0].replaceAll("\\
              ", "/");
10        // Update the nodeData with the file URL
11        myDiagram.model.setDataProperty(node.data, "fileURL", fileURL
              );
12        // Refresh the diagram
13        myDiagram.requestUpdate();
14  ...
15
16  doubleClick: (e, node) => {
17    console.log(node.data.fileURL);
18    if (node.data.fileURL !== null) {
```

```
19      fetch(node.data.fileURL)
20        .then((response) => response.json())
21        .then(
22          (jsonResponse) =>
23            (myDiagram.model = go.Model.fromJSON(jsonResponse))
24  ...
```

Listing 4.17: Implementation of linking transition to submodule

4.2.12 Tools

Our CPN editor offers several basic features that enhance the user experience and streamline the modeling process. One feature is the ability to undo and redo operations using keyboard shortcuts. The shortcuts for the undo and redo is the default Ctrl+Z and Ctrl+Y, respectively. The mouse wheel behaviour is set to zooming, in the modelling field, allowing them to adjust the level of detail in the models according to their preference. These essential undo, redo and mouse wheel behavior are set up when creating the diagram, as shown in Listing 4.18

```
1  myDiagram = $(go.Diagram, "myDiagramDiv", {
2    "undoManager.isEnabled": true, // enable undo and redo
3    "toolManager.mouseWheelBehavior": go.ToolManager.WheelZoom,
4  });
```

Listing 4.18: Modelling rules properties

To navigate the canvas, users can utilize click-and-drag functionality. Using this, users can pan around the model, providing a flexible and intuitive way to explore different parts of the diagram.

By clicking and holding, the user can create a selection box, enabling selection of multiple places, transitions, and arcs at once and perform operations on the selected elements.

Additionally, the GoJS incorporates common editing operations by default, such as copy, paste and cut, which further enhance the editing capabilities of the framework.

These features and interaction features contribute to a more efficient and user-friendly modelling experience.

4.2.13 Simulation

In order to integrate simulation functionality into our editing tool, we have developed a mock up of a simulator along with a visualization component that illustrates the changes made to a model during simulation. We will going forwards in this section refer to the mock up simulator as simulator. The simulator operates based on a set of states that the model can assume, where each state represents the state of each place and transition. An example of a state will be shown in Section4.3.1.

When a message is sent to the simulator, it responds with the next state of the model. Our software then iterates through each node in the model and updates

its state accordingly. For places, this involves modifying their marking, while for transitions, they may be highlighted based on whether they comply with the rules of a CPN model.

To interact with the simulator, we have implemented a HTML button with an `onClick` event that trigger the sending of messages to the simulator. In response, the simulator provides the next state of the model.

The simulator functions, as depicted in Listing 4.19, manage the states of the model. The states are stored as JavaScript objects in a list. The simulate function selects the appropriate state to be sent to the editor. The count variable keeps track of the number of states modeled, and once it reaches 6, it resets to 0.

The `testServerExecution` function is executed when the execute button is clicked. It receives the state from the simulator and sends it to the `sim` function.

The `sim` function processes each property in the state object and identifies the corresponding node in the model. If the node is a transition, it checks whether it should be highlighted and sets the highlight color accordingly. Otherwise, it sets the color to the default stroke, effectively disabling an enabled transition from the previous state. For places, the marking text is set to the appropriate message, ensuring the correct current marking is displayed.

This simulator functionality provides a glimpse into how the model would behave when executed, allowing users to visualize and analyze its dynamics. It serves as a valuable tool for testing and verifying the behavior of the model under different scenarios.

```
exports.simulate = function () {
  if (count === 6); count = 0;
  count++;
  return runningSequence[count-1];
};

function testServerExecution() {
  sim(mockserver.simulate());
}

function sim(m) {
  Object.entries(m).forEach(([nodeName, message]) => {
    myDiagram.nodes.each((n) => {
      nodeText = n.findObject("NODENAME");
      if (nodeText.text === nodeName) {
        if (n.category === "Transition") {
          if (message === "highlight") {
            n.findObject("NODESHAPE").stroke = enabledColour;
          } else {
            n.findObject("NODESHAPE").stroke = "#00A9C";
          }
        } else {
          n.findObject("MARKING").text = message;
        }
...
```

Listing 4.19: Function for loading the model

4.2.14 Saving and loading

Saving and loading the progression on models is an important part of modeling. We have introduced a different set of functionality, which we introduce in this Section.

The application includes a mechanism for listening to changes in the model, as shown in Listing 4.20. This functionality is implemented using GoJS `DiagramListener`, which monitors modifications made to the diagram. When a modification occurs, the listener locates the save button and enables the button by setting its `disabled` property to `false`, if found.

Next, the listener searches for the index of the `*` symbol in the document title and stores it in the variable `idx`. If the diagram has been modified and `idx` is less than 0, indicating that the `*` is not present in the title, the listener adds the `*` symbol to the document title, signifying that changes have been made. Conversely, if the diagram has not been modified and `idx` is greater or equal to 0, indicating that the `*` symbol is present in the title, the listener removes the `*` symbol from the title, indicating that there are now no unsaved changes.

This is implemented to ensure that the user is aware of any modifications made to the diagram and provides visual feedback on the save status, allowing them to decide when to save their changes.

```
myDiagram.addDiagramListener("Modified", (e) => {
  var button = document.getElementById("SaveButton");
  if (button) {
    button.disabled = !myDiagram.isModified;
  }
  var idx = document.title.indexOf("*");
  if (myDiagram.isModified) {
    if (idx < 0) {
      document.title += "*";
    }
  } else {
    if (idx >= 0) {
      document.title = document.title.slice(0, idx);
    }
  }
});
```

Listing 4.20: Listener for modifications in diagram

The saving and loading functionality in GoJS is demonstrated in several samples provided on their website. One common approach involves using an HTML `<textarea>` element to displayed the last save JSON representation of the model.

To implement saving, an HTML button is assigned an `onClick` event that triggers the saving function, shown in Listing 4.21. The save function retrieves the JSON representation of the model and sets it as the value of the `<textarea>` element located beneath the modelling canvas. Additionally, the function resets the `isModified` variable of the diagram to `false`, ensuring that any changes made to the previous model are cleared.

This makes it easy to save the current model by clicking the save button, updating the JSON representation.

```
function save() {
  document.getElementById("mySavedModel").value = myDiagram.model.
      toJson();
  myDiagram.isModified = false;
}
```

Listing 4.21: Function for saving the model

Loading, shown in Listing 4.22, complements the saving functionality by allowing users to load a model from the `<textarea>`.

To implement loading, an HTML button is assigned an `onClick` event that triggers the loading function. When the button is clicked, the function retrieves the text from the `<textarea>` element using the `getElementById` method. The text represents the JSON representation of the model. The function then updates the model by setting it to the JSON representation obtained from the `<textarea>`.

```
function load() {
  myDiagram.model = go.Model.fromJson(
    document.getElementById("mySavedModel").value
  );
}
```

Listing 4.22: Function for loading the model

By default the stored model will be in the source code of the HTML `textarea`. As we explained, the loading function only loads changes from the textarea. For a more functional loading we would rather load from a file. Having the possibility to load from a file also opens up the need for downloading the current JSON representation of the model.

4.2.15 Downloading and uploading

We have implemented the ability to download the model in our application. In order to do this, the user first needs to specify a filename using an input field and then click the download button.

Listing 4.23 demonstrates the downloading function-. To create the downloadable file, we generate an element and assign it the JSON representation of the model as its content. This JSON model, along with the filename is then passed to a download function. Within this function, the JSON model is converted into a blob with the MEME type `application/json`. By generating a file URL for this blob, we enable the user to download the file represented by the blob.

Next, we set the generated URL and the filename as attributes of the element. To ensure that the element remains hidden from the user, its visibility is set to `none`.

To initiate the file download, we add the element to the documents body and simulate a click on it. This action triggers the download of the file specified in the element. Consequently, we remove the element from the body, effectively cleaning up the DOM and completing the download process.

```
function downloadFile() {
  const filename = document.getElementById("filename").value;
  const content = document.getElementById("mySavedModel").value;

  if (filename && content) {
    download(filename, content);
  }
}

function download(filename, content) {
  const element = document.createElement("downloadElement");
  const blob = new Blob([content], { type: "application/json" });
  const fileURL = URL.createObjectURL(blob);

  element.setAttribute("href", fileURL); // file location
  element.setAttribute("download", filename); // filename
  element.style.visibility = "none";

  document.body.appendChild(element);
  element.click();
  document.body.removeChild(element);
}
```

Listing 4.23: The code for downloading functionality

The uploading and creation of the file tree is integrated with the tree view functionality as described in Section 4.2.10. To provide an overview, the process involves an HTML input element that accepts repositories as input. The user selects the repository containing their desired model. The file tree is then generated based on the files within the selected repository.

In our example, we will use the file structure for the two-phase commit protocol, which is outlined below:

```
TwoPhaseCommitProtocol
├── TwoPhaseCommitProtocol.json
└── Submodules
    ├── Coordinator.json
    └── Workers.json
```

The "TwoPhaseCommitProtocol" directory serves as the root director, containing the main model with a subdirectory containing the submodules of the two-phase commit protocol model.

An example of creating this sort of file tree from the model example is given in 6.1.

4.2.16 Palette

As GoJS explains it, "A palette is a subclass of Diagram that is used to display a number of Parts that can be dragged into the diagram that is being modified by the user"[25]. The palette component that facilitates the drag-and-drop functionality for adding places and transitions to the diagram. These places and transitions come preloaded with placeholder data in their connected inscription fields. The purpose of the placeholders is to visually indicate the location of the inscription fields and provide examples of how to utilize them. The palette itself comprises the following components, in the order of Figure 4.13:

- Placeholder - This element is used to place the frequently used places and transitions under the watermark from Northwood studios and GoJS.
- Transition - Representing a transition node, this component tends to become crowded due to its default small size and the inclusion of inscription fields within its border.
- Place - As a place node, this component showcases inscriptions as examples of how to label and annotate places within the model
- Group - Deprecated group object
- Comment - Although not previously mentioned, this component offers a means to add commentary or notes to specific elements within the model.

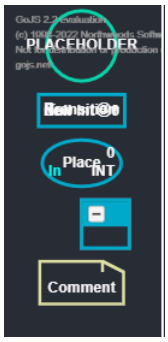


Figure 4.13: The palette in GoJS solution

Listing 4.24 shows us the implementation of the palette in our application. Palette is a separate diagram in GoJS and serves as a tool for users to drag and drop places and transitions onto the diagram.

To configure the behavior of the palette, we make some adjustments. Firstly, we disable the initial animation style, enabling us to use a custom animation called `animateFadeDown`.

The palette utilizes the same node templates as the main diagram, ensuring consistency in the appearance of the nodes. The deprecated group object is still included in the palette.

To define the contents of the palette, we set up various nodes in a specific order. First is the placeholder node, which serves as a visual reference for pushing frequently used transitions and places beneath the watermark. Following the placeholder, we include the transition and place nodes in the palette. These nodes can be dragged and dropped onto the main diagram, with the placeholders in the correct inscription fields.

Finally, we add group and comment nodes to the palette. The comment node provides a way to add annotations or notes.

The palette component enhances the usability of our application by providing an intuitive interface for adding nodes to the diagram. Users can easily select and drag the desired nodes from the palette.

```
myPalette = $(
  go.Palette,
  "myPaletteDiv",
  {
    "animationManager.initialAnimationStyle": go.AnimationManager.
        None,
    InitialAnimationStarting: animateFadeDown,

    groupTemplateMap: myDiagram.groupTemplateMap,
    nodeTemplateMap: myDiagram.nodeTemplateMap,
    model: new go.GraphLinksModel([
      { category: "Start", text: "PLACEHOLDER" },
      {
        category: "Transition",
        text: "Transition",
        guard: "[]",
        time: "@+",
        code: "new Int = 0",
        priority: "P_NORMAL",
      },
      {
        category: "Place",
        text: "Place",
        initMark: "0",
        unit: "INT",
        IO: "In",
      },
      { category: "Group", text: "Group1", isGroup: true },
      { category: "Comment", text: "Comment" },
    ]),
  }
);
```

Listing 4.24: Implementation of the palette

4.3 Code structure

4.3.1 Integration against the simulator

We have not yet started on implementing integration against the simulator. What we know now is that the modelling software uses JSON to represent the model and the simulator uses .cpn files as input, and outputs .fs(F#) files. Example data of what the simulator might send as a response to a request to simulate a step in the editor might be something like Listing 4.25.

To open the possibility of integrating the simulators F# output to the editors JSON representations, it is essential to incorporate a parser component, that is responsible for converting F# CPN files to JSON format. This parser plays a crucial role in enabling seamless data transfer and compability in a possible united system, ensuring efficient communication and accurate representation of simulation results within the editor environment.

At the same time, the mock simulator that the editing software so far supports lacks the support to read more than the highlighting of a transition and the current marking of a place. We can see an example of a state in Listing 4.26. One of the softwares would need to implement either a naming convention or

id convention. The preferred solution would probably be to create a parser component as mentioned earlier.

```
// Place
{ id = "ID1591819253"
name = "CoordinatorIdle"
colset = "UNIT"
initialMarking = Some "1^()" }

// Arcs
{ place = "ID1591819253"
transition = "ID1591819228"
expr = "1^()"
direction = PT }

// Transition
{ id = "ID1591819228"
name = "SendCanCommit"
guard = None; }
```

Listing 4.25: Information about a place, an arc and a transition

```
var ack = {
  "Worker\nIdle": "1`wrk(1)++\n1`wrk(2)",
  Acknowledge: "1`wrk(1)++\n1`wrk(2)",
  "Waiting\nAcknowledgements": "1`[wrk(1),wrk(2)]",
  "Receive\nAcknowledgement": "highlight",
  Decision: "",
  "Receive\nDecision": "",
  "Waiting\nDecision": "",
  "Collected\nVotes": "",
  "AllVotes\nCollected": "",
  "Waiting\nVotes": "",
  Votes: "",
  "Collect\nOneVote": "",
  CanCommit: "",
  "Receive\nCanCommit": "",
  "Coordinator\nIdle": "",
  SendCanCommit: "",
  CollectAllVotes: "",
};
```

Listing 4.26: Example state of from the mocksimulator

4.3.2 Canvases

The application has three different canvases; Modelling canvas, palette canvas and the tree view canvas. These canvases are placed in their own HTML div elements that links to the diagram, palette or tree view. The three HTML `div` elements, that are as shown in Listing 4.27, put inside their own ”editor” div

```
<div id="editor" style="width: 100%; height: 1000pt; display: flex;
     justify-content: space-between">
  <div>
    <div id="myPaletteDiv" style="height: 50%; width: 160px; margin
        -right: 2px; background-color: #282c34;">
    </div>
    <div id="myTreeDiv"
      style="height:50%; width: 160px;margin-bottom: 2px; margin-
          right: 2px; background-color: #282c34">
```

```
7       </div>
8     </div>
9     <div id="myDiagramDiv" style="flex-grow: 1; background-color:
          #282c34;"></div>
10  </div>
```

Listing 4.27: Link between HTML and GoJS canvases

The linking is done, so that the diagram referring to a div id. as in Listing 4.28, will be rendered in the referred div. In the Listing below, `myDiagramDiv` refers to `myDiagramDiv` in Listing 4.27.

```
1  function init() {
2    myDiagram = $(go.Diagram, "myDiagramDiv", {
3    ...
```

Listing 4.28: Link from GoJS to HTML

Chapter 5

Code Base

In this chapter we explain the structure of the code base in the project, providing insights into the organization and architecture of the software implementation.

5.1 File Structure

The following list shows the file collection used in developing the CPN editor.

A significant portion of the functionality residing in the `gojs.js` file. This is due to the interconnected objects and coding standard of GoJS. Especially the templates for nodes, links and the setup for the three canvases: modeling, palette and tree view.

1. index.js
2. index.html
3. gojs.js
4. grouping.js
5. filemanagement.js
6. contextmenu.js
7. mockserver.js
8. styles.css

The following sections provide insights into the structure of the listed files and how they connect to the overall architecture of the CPN editor application. Each file has a specific role and contributes to the cohesive functioning of the software. By understanding their relationships and roles, we gain an understanding of the software's architecture and design.

5.2 Electron implementation

The `index.js` file is responsible for initializing the Electron browser window for our application. In Listing 5.1, we can observe the step-by-step process of initializing the window. To ensure proper functionality, we import and obtain the necessary dependencies. Once the application is ready, it proceeds to create the `BrowserWindow`, which serves as the foundation for the entire application.

The `BrowserWindow` is instantiated with specific dimensions of 1600 pixels width and 400 pixels height. To facilitate integration with Node.js, we enable the `nodeIntegration` feature, while ensuring that `contextIsolation` is disabled. Additionally, we enable the `remoteModule` to support remote functionality within the application.

Subsequently, we load the `index.html` file as the initial action of the application. From this point forward, the development process aligns with standard web development practices, apart from one listener event: the window-closed event listener, which monitors for the closure of the window.

```
1  const electron = require("electron");
2  const app = electron.app;
3  const BrowserWindow = electron.BrowserWindow;
4  var mainWindow = null;
5  app.on("ready", () => createWindow());
6
7  function createWindow() {
8    mainWindow = new BrowserWindow({
9      width: 1600,
10     height: 400,
11     webPreferences: {
12       nodeIntegration: true,
13       contextIsolation: false,
14       enableRemoteModule: true,
15     },
16   });
17   mainWindow.loadURL(`file://${__dirname}/index.html`);
18   mainWindow.on("closed", () => {
19     mainWindow = null;
20   });
21 }
```

Listing 5.1: Opening a window using electronjs

Listing 5.2 demonstrates the functionality for quitting the application if all windows running is closed. This works for all the operating systems that Electron can run on, except if the operating system is based on Darwin, which typically refers to macOS. The reason for this distinction is that while Linux and Windows operating systems usually terminates an application when all windows are closed, macOS tends to continue the execution even without any open windows.

Since the development of the application primarily took place on a Windows platform, the macOS activation and related functionalities have not been extensively addressed. It is worth noting that further implementation details may be required to ensure optimal behavior on macOS, such as handling application activation and window management in accordance with macOS convention.

```
app.on("window-all-closed", function () {
  if (process.platform !== "darwin") {
    app.quit();
  }
});
```

Listing 5.2: Closing the application

5.3 Editor HTML page

In the `index.html` file, illustrated in Listing 5.3, we have set the layout of the page to accommodate the various components of our CPN editor. The page structure includes dedicated sections for the modeling canvas, palette, and tree view canvas, which are positioned at the top of the page. These sections provide the primary workspace for creating and manipulating CPN models.

Directly below these canvases, we have placed the applications buttons for saving and loading the model to and from the JSON representation, and also the button for executing the mock simulator.

The textarea, used for displaying the JSON representation of the model, is located just below the buttons and the modeling canvas. This placement ensures that it is easily accessible while working with the model.

The file management buttons, such as the download and load from files buttons, are positioned under the textarea. These buttons are not as frequently used as the core functionality buttons, but are still important for managing external files. Additionally, the button for creating the tree view is also placed within this section, allowing users to generate and visualize the file tree structure.

```

  <button id="SaveButton" onclick="save()">Save</button>
  <button onclick="load()">Load</button>
  <button id="printbutton" onclick="printData()">Print Data</button
      >
  <button id="execute" onclick="testServerExecution()">execute</
      button>
  <textarea id="mySavedModel" style="width: 100%;height: 300px;">

  </textarea>
  <input id="filename" placeholder="Specify a filename" />
  <button id="SavetoFileButton" onclick="downloadFile()">Download</
      button>
  <br />
  <input type="file" id="file" onchange="readFile(this)">
  <input type="file" id="fileInput" webkitdirectory multiple
      onchange="readDirectory(event)">
  <br />
  <button onClick="loadTree()">Show folders/files</button>
</body>
</html>
```

Listing 5.3: HTML snippet from index.html

5.4 The GoJS implementation

The `gojs.js` file is the largest file as stated earlier due to most of the functionality being defined here, using over 800 lines. The reason for collecting so much code in one file is to collect all GoJS at the same place. Some examples includes; defining most of the objects, as places, transitions and arcs. The size is mostly due to places, and transitions taking up much space for defining several TextBlocks for their inscriptions. Places and transition templates occupy 80-90 lines each.

Functionality associated with defining ports in the CPN editor amounts to a large code base. This is due to the functionality and shape of the ports.

In addition to the mentioned above definitions, the same file also handles the creation and template for the file tree.

Furthermore, the file plays a significant role in loading the process of JSON files into CPN models. Since the diagram object, which contains the model, communicates with this file, it facilitates the seamless integration of external JSON files into the editor.

5.5 Grouping

The deprecated functionality from the grouping feature was originally located in the `gojs.js` file, but after being deprecated it was moved to `grouping.js`. The functionality includes setting layout, adding properties to the group template, making sure the group has the correct ruleset and behaviour. Functionality for adding places and transitions to the group by dragging and dropping. At last there is highlighting for when holding a place or transition above the group, indicating the node will be placed in the group if dropped at that moment.

5.6 File management

The `filemanagement.js` file in the CPN editor project serves as a crucial component for the addition of various functionality related to file handling. The implementation include saving files, downloading files, reading directories and generating the file tree. See Section 4.2.15 for a more in-depth explanation.

5.7 Context menu

The `contextmenu.js` file comprehensively handles all aspects of the context menu's implementation. As explained in Section 4.2.9, it encompasses the inclusion of various buttons within the two context menus. Also functionality for changing the colours on transitions and places, and arrowheads to change the type of arc.

We have also started created some functionality towards linking file URLs to transitions, and use this towards easier and better functionality concerning substitution transitions.

5.8 Mockserver

`Mockserver.js` contains the 6 necessary states for going through a complete commit run. We have put all of these in a list and lastly there is a function for sending the correct state to the application. Section 4.3.1 presents more information on this file.

Chapter 6

Evaluation

In this chapter, we look at the different features of CPN Tools and our application, comparing them and assessing their functionality, performance and usability among others.

6.1 Creating a CPN Model

When initiating the creation of a CPN model using our new editor, it is beneficial to begin by outlining the desired model structure. As an illustrative example, we consider the two-phase commit protocol. Initially, our focus is on constructing the complete model without incorporating any abstractions. This approach allows us to gain a comprehensive understanding of the model's interconnections before introducing any higher-level conceptualization.

The initial step involves positioning the initial places and transitions in the model, placing them approximately in their respective locations. As part of this step, we assign meaningful names to each place and transition and resize them to accommodate future inscriptions. At this stage, our model takes shape, resembling the structure depicted in Figure 6.1, consisting of a total of 9 places and 6 transitions positioned within the model.

We proceed to establish connections between our places and transitions by introducing arcs between them. This process, as detailed in Section 4.2.6, involves hovering the mouse over one of the four sides of a place or transition, dragging the arc to the desired port of a node of the opposing type, and finalizing the connection by releasing the left mouse button. By following this procedure, our model begins to take shape, resembling the structure depicted in Figure 6.2.

We proceed to add place and transition inscriptions to our model. In the case of the two-phase commit protocol, we begin with the `Coordinator Idle` place, where we set the initial state to 1. For the `CanCommit` place, we need to specify the colour set type as `Worker` by entering this information in the lower right corner inscription field of the place. The `Worker Idle` place is assigned an initial marking of `Worker.all()` and colour set of `Worker`. By incorporating these inscriptions, we indicate that in the initial state, the `Worker Idle` place will

Figure 6.1: Current state of the model after adding places and transitions

contain, by default, both workers represented as `1‘wrk(1)++1‘wrk(2)` during simulation.

For the `Collected Votes` place, we set the colour set as `WorkerxVotes` and the initial mark is assigned to an empty list, denoted as `[]`, representing no collected votes initially. As for the `AllVotes Collected` place, it requires a guard expression, and in this model, the only guard expression used is `[All Votes]`.

Moving on to the two lowest places in our model, the `Decision` place is assigned the colour set `WorkerxDecision`, indicating that it represents decisions made by the workers. The `Acknowledge` place, on the other hand, has the colour set `Worker`.

Lastly, we have the `Waiting Acknowledgements` place, which is assigned with the colour set `Workers`, indicating the wait from multiple workers.

In our example model, we have assigned arbitrary timed values to represent the duration or execution time these transitions. Specifically, the transitions `Receive CanCommit`, `Receive Decision` and `Receive Acknowledgement` have been assigned the respective timed values of `@+5`, `@+3` and `@+4`.

The final step before having a complete two-phase commit protocol without any abstraction is adding arc inscriptions. This involves assigning inscriptions to the arcs connecting different places and transitions. In our case, we have several arc inscriptions to add.

Starting from `CanCommit` and `Worker Idle` going into `Receive CanCommit`, both arcs have the inscription `w`. Moving on to `Receive CanCommit`, there are three outgoing arcs. Two of these arcs have the inscriptions specified in Listing 6.1. These inscriptions determine the behavior of the arcs based on the vote received. If the vote is Yes, the arc sends `1‘w` to `Waiting Decision`. On

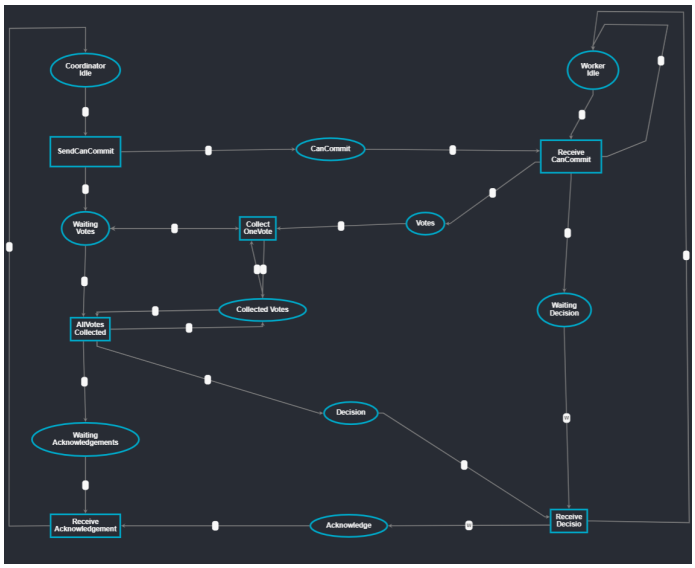


Figure 6.2: State of model after adding arcs

the other hand, if the vote is No, the arc sends `1‘w` to `Worker Idle`. These arc inscriptions define the flow of the model and the actions performed based on specific conditions.

```
// To Waiting Decision
if vote = Yes
then 1‘w
else empty

// To Worker Idle
if vote = No
then 1‘w
else empty
```

Listing 6.1: Arc inscription from Receive CanCommit to Waiting Decision and Worker Idle respectively

The remaining arc from `Receive CanCommit` is responsible for sending `(w, vote)` to `Votes`. The next arc from `Votes` contains the same inscription. Moving on to the `Collect OneVote` and `Collected Votes`, these two places are connected by two arcs. One arc goes from `Collect OneVote` to `Collected Votes`, while the other arc goes in the opposite direction. From `Collect OneVote` we send the function `Addvote` as the arc inscription. This is a function which is in the CPN Tools version of the model, defined as shown in Listing 6.2. The other arc, returning from `Collected Votes` is inscribed with `votes`.

From `Collected Votes` there is an arc going to `AllVotes Collected` with inscription `votes`. `AllVotes Collected` returns an empty list to `Collected Votes`, indicated using another arc with inscription `[]`. With these inscriptions, we have completed the Collected votes path of the two-phase commit protocol, defining the flow and actions associated with the collection of

votes.

```
1 // Collect OneVote to Collected Votes arc inscription
2 Addvote ((w,vote),votes)
3
4 // Definition of Addvote function
5 fun AddVote ((w,vote),votes) = (w, vote)::votes;
```

Listing 6.2: Addvote definition and inscription

In the final part of the model, we have a few more arc inscriptions to complete. Starting with the out arcs from `AllVotes Collected`, there are two destinations: `Decision` and `Waiting Acknowledgement`. The arc inscription to `Waiting Acknowledgement` is `YesWorkers votes`. This indicates that the arc carries the information of the workers voting yes to the next place.

Moving on to the arc connecting `AllVotes Collected` and `Decision`, the inscription `InformYesWorkers votes`. This function will decide if the workers voting yes, should receive a commit or abort message as a response from the coordinator.

Next, from `Decision` to `Receive Decision`, the arc inscription is `w, decision`. For the three other arcs connected to `Receive Decision`, their inscriptions are all `w`. Sending workers to the connecting places.

Finally, the two input arcs to `Receive Acknowledgements`, have their own inscriptions. The arc coming from `Waiting Acknowledgement` is inscribed with `workers`. On the other arc coming from `Acknowledge` has the inscription `list_to_ms workers` after being converted into a list.

6.1.1 Abstraction

Creating substitution transitions in the editor can still be a somewhat delicate process. One approach is to selectively delete selections of the model and download the remaining part as separate models.

To begin, it is important to save the current progress by clicking on the save button. Then, the model can be downloaded by assigning it a suitable name, such as "2FCP", and selecting the download button. It is recommended to create a dedicated directory to store the downloaded "2FCP.json" file and move it to this directory. Additionally, a second directory named "submodules" needs to be created in the same location.

To create the Coordinator submodule, the Worker section of the model needs to be deleted, resulting in a structure similar to Figure 6.3. After making these modifications, pressing the save button and download the Coordinator submodule with an appropriate name, such as "Coordinator". To ensure correct file hierarchy, we download the file to the submodule directory.

To create the Worker submodule as shown in Figure 6.5, we start by loading the full model from its file. Then, we proceed to delete all the objects making up the Coordinator submodule, while ensuring to retain the connections between the two substitution transitions, which are `CanCommit`, `Votes`, `Decision`, and `Acknowledge` places. It is important to rename the In/Out sockets correctly, with `CanCommit` and `Decision` as In sockets, and `Votes` and `Decision` as Out

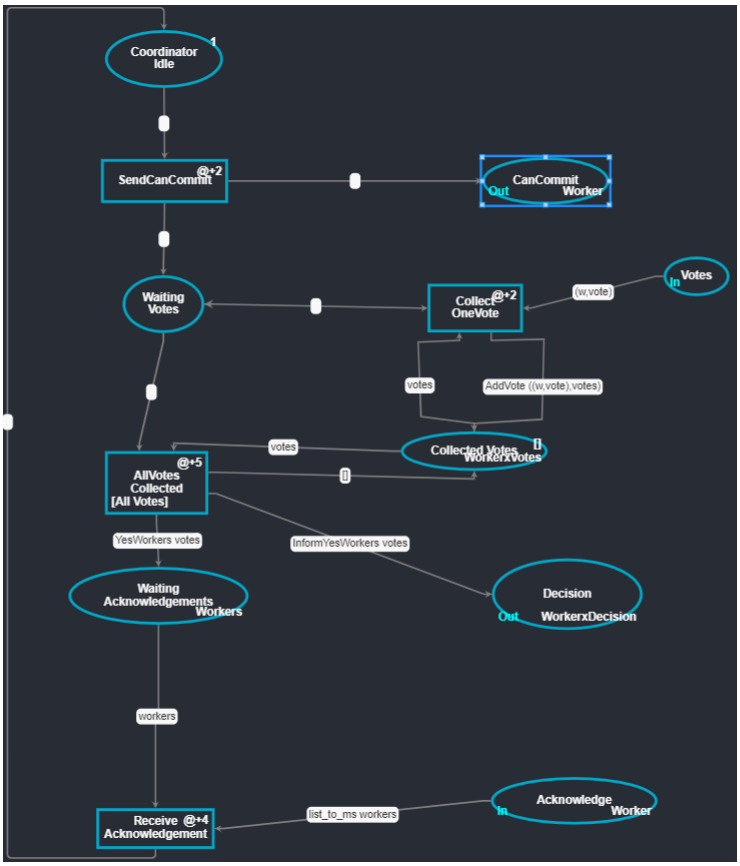


Figure 6.3: Coordinator submodule in new editor

sockets. Once these modifications are made, we save and download the submodule as explained previously. The file will be named "Workers" and downloaded into the same submodules directory as the Coordinator submodule.

The final step involves creating an overview of the model. We once again load the main model file, "2FCP.json", and delete all the places and transitions, except for the socket places: `CanCommit`, `Votes`, `Decision`, and `Acknowledge`. Subsequently, we create two new transitions that will be used to represent the submodules. We place one on the left, with the name `Coordinator`, while the other is placed on the right and called `Workers`. The socket inscriptions on the two, lowest socket places, `Decision` and `Acknowledge`, should be removed. The two highest places, `CanCommit` and `Votes`, should retain their In/Out inscription. At this point, our model should resemble Figure 6.4. To link the transitions to their submodules, right-click the transitions to open the context menu, click `link submodule` and select the corresponding file of the submodule. We can save and download this final model, overwriting the previous version. It is important to note that there is a bug where the text may appear cluttered when reloading this model. For more information on this bug, see Section 7.3.1 and Figure 7.2.

Figure 6.4: The two-phase commit protocol in our application with abstraction

We have now modelled and saved the entire two-phase commit protocol with abstraction, the model directory, should now look something like the directory tree as seen in Section 4.2.16, when loading the directory.

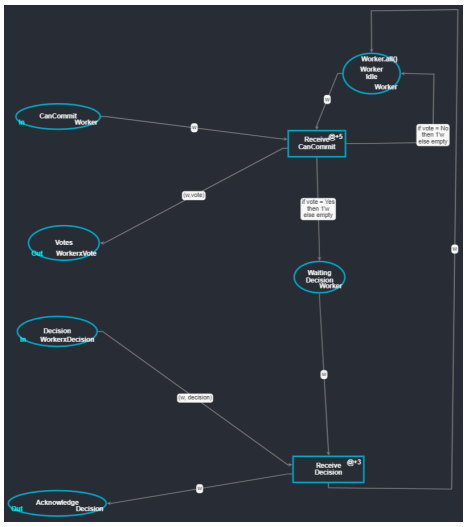


Figure 6.5: Workers submodule in new editor

6.2 Comparison to CPN Tools

In this section we will compare our new editor to the CPN Tools. We will refer to our project as the editor or application. We will refer to Table 2.1 from Section 2, which outlines the various requirements and the extent to which we have successfully addressed them. By examining these requirements, we aim to assess the comparative strengths and limitations of our editor in relation to CPN Tools.

6.2.1 Feature comparison

The layout of the editor has been redesigned to enhance the user experience and provide a more comprehensive modeling overview. Several notable changes have

been implemented, including changes in features such as downloading, saving, loading functionality. Additionally, certain features like editing the style of of places and transitions, as well as changing arc types, have been relocated to context menus for improved accessibility and ease of use. These modifications aim to streamline the modeling process and empower users with enhanced control and navigation options within the editor.

One of the reasons for improved overview in the editor is the removal of the simulator component. As previously mentioned, the focus of the current version is solely on providing a comprehensive editing environment. As an temporary replacement, we have included a mock simulator. In addition, the editor still lacks some functionality that has been given lower priority.

CPN Tools offers a comprehensive toolbox in their layout, containing 11 binders that provide various tools for editing, simulation, state space analysis, and other common editing operations. In the following sections, we explore and compare specific aspects of the toolbox along with other features and compare them to the our application and how well we fulfill our requirements.

6.2.2 Places, transitions and arcs

In this subsection, we compare the places, transitions and their associated features in CPN Tools and our application.

CPN Tools provides a "Create" binder, as depicted in Figure 6.6, which includes transitions, places, and three types of arcs; normal, inhibitor and reset arcs. Additionally, CPN Tools offers tools for deletion, cloning, changing arc direction, and horizontal and vertical magnetic line.

In our application, we have implemented a palette as a replacement for the toolbox. The palette contains the main CPN objects of places and transitions, conveniently located on the left-hand side of the modeling canvas.

Unlike CPN Tools, where each arch type has a dedicated tool for placing it, our application simplifies the process by allowing users to drag and drop arcs between places and transitions. Once an arc is places, we provide the option to edit its direction or type, through the context menu, accessible by right-clicking on the desired arc.

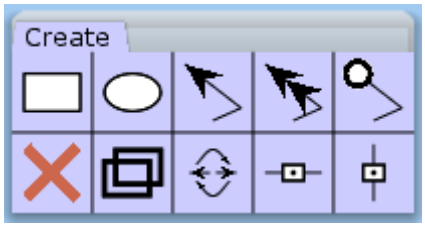


Figure 6.6: Create binder in CPN Tools

Not taking style into account, places and transitions in CPN Tools and our editor is similar from a front-end perspective. However, the main difference lies in how arcs are connected to places and transitions.

In CPN Tools, arcs are directly connected to the shape of the places and transitions. The arcs visually intersect or attach to the shape of the node itself. In contrast, our editor has introduced a different approach to connecting arcs. We use ports that are placed on the places and transitions. These ports become visible when hovering the cursor over the corresponding location on the node. Each place and transition in our editor has ports on all sides of their shape, providing multiple connection points for arcs.

This port-based approach is a practical solution for handling arcs in our editor. However, it should be noted that when multiple arcs are connected to the same port, particularly on places, they may overlap each other, potentially affecting the visual clarity of the model.

Places and transitions provide the necessary functionality for visualizing simulations and incorporating the required inscriptions fields. These nodes effectively represents the essential components of the Coloured Petri net models, allowing for the representation and manipulation of data and events throughout the model. They successfully meet the requirements outlined in R1 and R2.

6.2.3 Moving objects

In CPN Tools, nodes can be freely moved around the canvas, and the magnetic grid provides a helpful guide by snapping places and transitions to the gridlines. The feature assists in aligning nodes horizontally or vertically with each other, ensuring better visual organization. On the other hand, our application utilizes a default grid net in GoJS with a size of 10x10 pixels, which has been determined as the optimal size through testing different size grids. This grid net serves a similar purpose to CPN Tools' magnetic grid, aiding in aligning nodes on the canvas. Without this grid, it would be more challenging to achieve precise horizontal and vertical alignment of nodes in our editor.

6.2.4 Binders

In CPN Tools, the concept of binders allows users to open and view multiple models simultaneously, facilitating side-by-side comparison and analysis. On the other hand, our application takes a different approach by providing a single modeling canvas and utilizing a file tree view for managing multiple models. This design choice allows users to quickly switch between different models without the need for separate windows or complex menu navigation. We prioritize simplicity and ease of use, enabling users to rapidly access and edit the desired model without unnecessary distractions.

6.2.5 Style

In CPN Tools, the style tool offers users the ability to customize the visual appearance of places, transitions, and arcs, see Figure 6.7. This includes changing the color outline, line width arc head size, and toggle filling of elements. The style can also be cloned from one object to another. In our editor, we have simplified the process by allowing users to right-click on places and transitions to select the desired color from a context menu. Additionally, users can easily return the style to its default setting. However, we have not implemented the

option to change the line width of elements in our editor. To clone the style of an object, users can utilize the built in functionality of GoJS by holding down the CTRL key and clicking on the object they wish to clone, then dragging it to the desired location.

Figure 6.7: Style binder in CPN Tools

6.2.6 Functions and variables

In CPN Tools, users have the capability to define their own functions and variables in a dedicated field located in the sidebar, below the toolbox. This allows users to input custom code when loading or creating a model. However, in our current application, we have not yet implemented a separate code input field, apart from the code input inscription fields on the transition themselves. The inclusion of a code input feature is something that our application could consider in future updates, particularly if there is a plan to merge the CPN Simulator functionality with our application. While it may not be explicitly mentioned as a requirement, it can be viewed as a subrequirement that aligns with R6 - providing advanced modeling capabilities.

6.2.7 Inscriptions

Inscriptions in CPN Tools and our editor share similar functionality. In our application, inscriptions are placed within the nodes, which due to some limitations within the GoJS framework. However, during the late stages of this thesis, a potential solution was discovered. It involves creating a selection adornment for the inscription field to be positioned "outside" the node.

In our application, inscriptions are placed in the same manner as in CPN Tools. However, it is important to note that there is currently no direct connection between our editor and a simulator. Therefore, we rely on state changes from a simulator to provide functionality for our inscriptions.

The process of selecting inscriptions has undergone some changes as well. While CPN Tools allows for toggling between different inscriptions using the tabular button, in our application, the user needs to click on the location of the inscription fields to select them. This may pose a challenge for new users, particularly since the inscription fields are initially invisible when there is no text present. To address this, we utilized placeholders when placing new places and transitions to provide guidance to the user.

6.2.8 Markings

In CPN Tools, the current marking is visually represented by a green circle, indicating the count of elements in the marking, along with a block of text displaying the specific markings. However, in our editor, we have chosen a more minimalist approach for the current marking. This is primarily due to the issue of text cluttering within the place node, as the current marking overlaps within the name of the place it is located on.

While our current implementation serves as a prototype and fulfills the requirement R3, there is room for improvement to enhance readability. This would ultimately make the editor more user-friendly. The solution for improving both the current marking and addressing the concerns mentioned in Section 6.2.7 would be the same.

6.2.9 Arcs

In our application, we have implemented ports as the connection points for arcs, as discussed in Section 4.2.7. Unlike CPN Tools, where the placement of arc inscription field follows a more intricate logic to avoid overlapping with other objects in the model, we have a semi-fixed arc inscription field. Additionally, our arcs follow orthogonal pathing to leverage the predefined pathing functionality provided by GoJS, specifically the `AvoidNodes` option. This choice ensures that when moving or resizing arcs, their pathing will not overlap any nodes in the model.

Another distinction between CPN Tools and our application is that we have a single type of arc, but allow for customization by changing the arrowhead style. Moreover, arcs in our application are added by dragging from ports, as opposed to placing them as separate objects in CPN Tools.

Considering the differences between our application and CPN Tools in terms of how arcs are handled, we can state that we fulfill the requirement R4 that we established for arcs.

6.2.10 Substitution transitions

The use of substitution transitions is significant in large CPN models as they provide a means of abstracting the model, making it easier to introduce, edit, and use in general. Fulfilling this requirement was important during the development of our editor.

During the development process, we explored two potential solutions for implementing substitution transitions: utilizing GoJS groups and the current approach of creating separate files and models for submodules (see Section 4.2.11).

Initially, we considered using GoJS groups as they visually resembled substitution transitions in CPN Tools. However, we encountered several challenges with this approach. When opening groups, the placement of nodes would not be preserved, resulting in a cluttered arrangement of places and transitions. Additionally, the arcs between nodes would take inefficient paths. Moreover, achieving the desired level of abstraction proved difficult using this solution. As

a result, the decision was made to deprecate grouping for substitution transitions.

Instead, we developed a solution that involved creating separate files for each submodule, allowing for easy navigation between them using the tree view feature discussed in Section 4.2.10, but also the addition of linking transitions to submodules using the context menu. While there is still room for further refinement, the current solution provides a high level of abstraction and functionality, fulfilling R5.

This approach parallels the concept of substitution transitions in CPN Tools. While using binder tabs to view only the substitution transitions is a potentially useful feature, it does not serve as a functional requirement.

6.2.11 Runtime environment

Requirement R6 in Table 2.1 stated the need for a runtime environment to run our application. This requirement aims to provide users with a baseline for modeling CPN models. We conclude in the following subsections on how we fulfill the requirement for a runtime environment.

Canvas

The need for a canvas to facilitate the the modeling process was identified early on in our project. We ended up creating three distinct canvases to cater to different functionalities required by our application. These canvases are the modeling canvas, the palette canvas, and the tree view canvas, as discussed in Chapter 4.

The modeling canvas serves as the primary workspace for users to create and modify their CPN models. It is the only canvas where users can directly edit the model by placing and connecting nodes. The palette canvas, on the other hand, functions as a repository of nodes that the users can drag and drop onto the modeling canvas to build their models. Lastly, the tree view canvas provides a hierarchical representation of the models and allows users to switch between different models by selecting the corresponding file.

In CPN Tools, a binder system is used, where a large canvas acts as the background, and users can open multiple binders to work with different models or modules. Each binder can contain a single model, and users can navigate between submodules and supermodules using binder tabs.

While the binder system in CPN Tools provides a flexible way to manage multiple models, we decided to follow a more traditional approach of having a single modeling canvas in our application. The tree view serves as a substitute for the binder tabs, allowing users to navigate between different models and view their hierarchical structure.

While implementing a tab system could potentially enhance the user experience, the presence of the tree view adequately fulfills the purpose of managing the modules of a model and the hierarchy. Thus, we believe that the decision to use a single modeling canvas was complemented by the tree view offers a suitable alternative to the binder system found in CPN Tools.

Buttons

During the development process, we recognize the need for additional features and incorporated them into our application by introducing new buttons to trigger these functions. The requirement R7 involves saving and loading functionality. Initially, we implemented saving and loading buttons to facilitate testing and development. These buttons primarily interacts with the textarea, enabling the saving of the model from the canvas to the textarea and loading a model from the textarea to the modeling canvas. While this satisfies the requirements to some extent, as it allows internal saving and loading within the application, it falls short of the requirements to save and load from files.

To address the saving functionality, we implemented a file download feature. When the the user downloads a file, the application creates a file containing the contents of the textarea, thereby achieving the saving to a file functionality.

For loading from a file, we provide two options. The first option is to choose a single file to load, which updates the model in the canvas with the contents of the selected file. The second option is to load a directory containing a CPN model. In this case, we also generate a corresponding tree view that displays the model's hierarchy. The user can then switch between different models by right-clicking on the files in the tree view.

In CPN Tools, there is also two different ways to load a model. Users can either open the radial menu by right-clicking on the background canvas, where the option to load a net is available. Alternatively, users can open the Net tool binder, which provides the functionality to load and save a net.

While we could enhance the saving functionality by automatically saving to files, the current functionality of manual saving and loading is still present. With a functioning loading mechanism and the ability to save models to files, we are confident that we have fulfilled the requirements stated in R7.

Extra features

As an additional feature in our application, we introduced a textarea located below the model canvas, as depicted in Figure 6.8. This textarea serves the purpose of displaying the JSON representation of the model.

The textarea provides an enhanced view of the model structure and data. By presenting the JSON representation, users can easily understand and analyze the various components of the CPN model. Furthermore, the textarea allows users to make modifications to the model if desired.

The presence of the textarea offers an alternative way to load and edit a model without relying solely loadign a function from a file. Users can directly input or modify the JSON representation in the textarea, providing flexibility and convenience.

To facilitate some interaction with the model in the textarea, we have included save and load buttons. The save button enables users to save the content of the model to a file, while the load button loads the model from the textarea into the canvas. We have also placed the execution button here, which executes the mock simulator to visualize state changes in the two-phase commit protocol.

Figure 6.8: A view of the textarea with JSON representation of a two-phase commit protocol model

Listing 6.3 presents a snippet of the JSON representation of the two-phase commit protocol, showcasing some key aspects of the protocol's structure and attributes.

```
{ "class": "GraphLinksModel",
    "linkFromPortIdProperty": "fromPort",
    "linkToPortIdProperty": "toPort",
    "nodeDataArray": [
  {"category":"Place","text":"Coordinator\nIdle","initMark":"1","key":-3,"loc":"-250 -440"},
  {"category":"Place","text":"Waiting\nVotes","initMark":"","key":-4,"loc":"-250 -110"},
  {"category":"Transition","text":"SendCanCommit","guard":"","time":"@+2","code":"","priority":"","key":-6,"loc":"-250 -270"},
  {"category":"Place","text":"CanCommit","initMark":"","unit":"Worker","IO":"In","key":-5,"loc":"261.83865649201147 -275"},
  {"category":"Transition","text":"AllVotes\nCollected","guard":"","time":"","code":"","priority":"[All Votes]","key":-2,"loc":"-230 100"},
  {"category":"Place","text":"Votes","initMark":"","unit":"WorkerxVote","IO":"Out","key":-7,"loc":"460 -120"},
  {"category":"Place","text":"Decision","initMark":"","unit":"WorkerxDecision","IO":"","key":-8,"loc":"304.43259997700534 275.0847030462221"},
  {"category":"Place","text":"Waiting\nAcknowledgements","initMark":"","unit":"Workers","IO":"","key":-9,"loc":"-250 330"},
  {"category":"Transition","text":"Receive\nAcknowledgement","guard":"","time":"@+4","code":"","priority":"","key":-10,"loc":"-250 510"},
  {"category":"Place","text":"Acknowledge","initMark":"","unit":"","IO":"","key":-11,"loc":"300 510"},
  {"category":"Place","text":"Worker\nIdle","initMark":"Worker.all()","unit":"Worker","IO":"","key":-14,"loc":"805 -435"},
  {"category":"Transition","text":"Receive\nCanCommit","guard":"","time":"@+5","code":"","priority":"","key":-15,"loc":"750 -250"},
  {"category":"Place","text":"Waiting\nDecision","initMark":"","unit":"Worker","IO":"","key":-16,"loc":"750 60"},
  {"category":"Transition","text":"Receive\nDecision","guard":"","time":"@+3","code":"","priority":"","key":-17,"loc":"760 500"},
  {"category":"Place","text":"Collected Votes","initMark":"[]","unit":"WorkerxVotes","IO":"","key":-18,"loc":"120 60"},
  {"category":"Transition","text":"Collect\nOneVote","guard":"","time":"@+2","code":"","priority":"","key":-19,"loc":"120.65443420410156 -115"}
  ],
```

Listing 6.3: partial JSON representation of the two-phase commit protocol

6.2.12 Modularity

In accordance with our requirement R8 for increased modularity, we have increased modularity, we have successfully achieved a higher level of modularity in our CPN editor compared to CPN Tools. While CPN Tools uses a single file for an entire model or net, our application takes a different approach by splitting the different substitution transitions into individual files. This modularity structure offers several advantages.

Firstly, it facilitates collaborative work on the same model, as different individuals can work on separate submodules simultaneously. This enables concurrent development and reduces conflicts when merging changes.

Secondly, the modular design allows for easier refactoring and making significant changes to submodules without impacting the entire model. Each submodule

can be modified independently, providing flexibility and maintainability.

By adopting this approach, we have successfully achieved our goal of increasing modularity in the CPN editor. This modular design not only enhances collaboration and flexibility but also fulfills our requirement R8 for a more modular editor compared to CPN Tools.

6.2.13 Common editing operations

Our CPN editor includes several common editing operations that are considered standard in modern software applications. These operations include copy, cut, paste, undo, and redo. These functionalities were readily available in GoJS framework, which we utilized for our editor. By incorporating these features, we aimed to provide users with familiar and intuitive editing capabilities.

The inclusion of keyboard shortcuts for these operations allows users to perform tasks more efficiently, enhancing their productivity while working with the editor. The availability of these shortcuts aligns with our requirement R9 for common editing operations

Furthermore, we also considered enhancing the user experience by providing context menus that include these common editing operations. Making it easier for users to discover and utilize these functionalities when interacting with the model elements.

6.3 Grading requirements

We have graded the requirements from low to high based on how we fulfill the requirements in Table 6.1. We have based the scored on functionality and visual functionality and ease of use.

ID	Name	Fulfillment
R1	Places	High
R2	Transitions	High
R3	Markings	High
R4	Arcs	High
R5	Substitution transitions	Medium
R6	Runtime environment	Medium
R7	Saving and loading	Medium
R8	Modularity	High
R9	Common editing operations	High

Table 6.1: Grading the results of requirements

Concerning the requirements with medium fulfillment, we discuss the reasoning for this.

Substitution transitions: Currently, there is no visual indicator to distinguish when a transition is connected to a submodule. Enhancements can be made to establish clearer links between substitution transitions and submodules, providing improved visual feedback to users.

Runtime environment - The application exhibits slow startup times and may require a manual refresh to load the implemented test model. Further refinement of the runtime environment's design and performance can contribute to a more seamless user experience.

Saving and loading - The current implementation of saving and loading functionality involves manual synchronization between the JSON representation and the model. This process can be perceived as tedious. Enhancements can be made to streamline the process by directly working with files, simplifying the interaction and eliminating the need for manual synchronization.

The proposed improvement for the requirements mentioned above are discussed in Section 7.3.

Chapter 7

Conclusion and Future Work

We conclude this thesis with summarizing the evidence and findings we have discovered through this thesis. The technologies we have used have shown to be suitable for the CPN editor we have developed.

7.1 Conclusion

These were the main questions we intended to investigate and answer at the start of this thesis.

- RQ1 What specific functionalities are essential for a CPN editor to effectively support modeling and simulation activities?
- RQ2 What are the strengths and limitations of various candidate software technologies in meeting the requirements identified in RQ1?
- RQ3 To what extent does our developed prototype demonstrate the necessary functionality and usability required for an effective CPN editor, as identified in RQ1?

In Chapter 2, we conducted an analysis of CPN and its features, and identified the important features of CPN modeling and simulation. We also identified the need for improved modularity to support hierarchical in the context of modern software, particularly for distributed systems. The requirements summarized in Table 2.1 and discussed in Section 2.3 outlined the essential features and their significance in a modern CPN editing tool. This addresses RQ1.

In Chapter 3, we explored various technology platforms for implementing the CPN editor. We evaluated Eclipse EMF Core, Eclipse GLSP, Elm programming language, GoJS, and lastly Electron for the runtime environment. After careful consideration, we determined that GoJS was suitable technology platform for developing a modern CPN editor, answering RQ2 when summarizing the frameworks in Section 3.7.

GoJS was our choice due to the following: The framework is relatively modern and was first released in 2012, and has since been updated regularly and is still currently updated and developed. It also has solid documentation with online API that provides a good search functionality. It is easy to maintain, due to being highly modular. The framework makes it easy to introduce features without disrupting the flow of the software. The community is still highly active, both on Stack Overflow, being the most popular community forum in software development, but also the Northwoods software support forum.

Chapters 4-6 detailed the development of the CPN editor throughout the thesis. We discussed the implementation of various features and functionalities, addressing the identified requirements from RQ1 as well as other necessary components for the application. We examined the underlying code structure and compared the implemented features with those in CPN Tools. The completion of a fully usable CPN model, such as the two-phase commit protocol, demonstrated the effectiveness of the developed editor. Although some small issues concerning substitution transitions, the overall functionality is robust. Thus, it is reasonable to conclude that the CPN editor would yield similar results with other CPN models as well, addressing RQ3.

7.2 Related Work

In this section, we provide an overview of existing tools for handling CPNs.

CPN IDE[7] is a tool that serves as a Java-based port of CPN Tools. It utilizes a JavaScript-based editor, and builds the simulating upon the capabilities of Access/CPN[30]. Access/CPN, a framework designed for extending analysis capabilities and integrating CPN models into external applications. To facilitate the communication between the editor and Access/CPN, CPN IDE employs a REST interface. This interface is implemented through a Java-based controller, which runs as a Spring Boot[27] server. Spring Boot, a popular Java framework, simplifies the development process by providing opinionated defaults and convenient auto-configuration capabilities.

Renew - The Reference Net Workshop[22] is a Java-based tool that offers a versatile environment for modeling and simulating systems using reference nets. It is designed to run on any machine equipped with a Java virtual machine. One features notable in Renew is its support for multi-formalism modeling. This allows users to create and manipulate reference nets, which provide a mechanism for interconnecting nets through synchronous channels. This capability enables concurrency and simulating multi-processor architectures.

CosyVerif[6] is a software environment designed for the formal specification and verification of dynamic systems. It provides a common framework that integrates various existing tools for specification and verification[1], aiming to streamline the process of analyzing system behaviour and properties. CosyVerif is not primarily focused on CPNs, but does offer support for CPNs to some extent. CPNs can be used as one of the formalisms within the framework, allowing users to create CPN models and perform analysis on them.

The ePNK[9] framework is a platform specifically designed for developing Petri

net tools based on the Petri Net Markup Language (PNML) transfer format. Its primary focus area is to facilitate the definition of different Petri net types, which can then be used within a GMF editor for visually editing nets of the corresponding type. There is thereby also a focus on high-level Petri nets.

Various other frameworks for editing CPNs, Petri nets, or other variations of Petri nets, are GCPN[14], MEdit4CEP-CPN[2], GreatSPN[11]. We have not investigated these further, due to their difference to CPN Tools and our CPN editor.

The difference between our CPN editor and other existing CPN tools primarily lies in the choice of frameworks and their specific areas of focus. While our editor utilizes GoJS as its framework and emphasizes the creation of a comprehensive editing platform for Coloured Petri Net, with future possibility of connecting a CPN simulator. Other tools may prioritize different types of Petri nets or emphasize specific analysis techniques.

7.3 Future Work

In this section, we discuss features of the project that have the potential for improvement and further development, with the aim to enhancing its functionality and user experience. By identifying areas for improvement, we can explore potential opportunities for future enhancements and advancements.

The CPN editor we have developed would have more finished features and simulation if a simulator is to be implemented as part of future work. The following subsections explain the most prominent bugs appearing when modeling, and other features and functionalities that could do with further development as part of the future work.

This project has not yet looked into taking .cpn files as input for the model. This is due to the possibility of developing a conversion tool for converting JSON files, being the default model representation of GoJS models, to .cpn for the use in CPN simulating tools and related to CPN Tools.

7.3.1 Places and transitions

There is for the time being a bug where the size of places and transitions reset to fit only the size of their name when restarting, or loading the project. This leads to clutter with text if the place or transition has inscriptions as seen in figure 7.1. Preferably, these inscriptions would be moved outside the place or transition itself, or outside the shape of the transition or place. As a minimum, the size of the places and transitions should be saved.

7.3.2 Arcs

Figure 7.2 shows a bug appearing with arcs are when loading a model or starting the software, some arcs are not fully connecting to their places and transition, this is related to the bug discussed in Section 7.3.1. To fix this, the user would have to move or resize the place or transition for the arc to snap to its port. This leads to the arcs repathing to their connected places and transitions, leading to

Figure 7.1: Transition in palette with all inscription fields used

another bug discussed below.

The bug is correlated to automatic pathing, the model does not save breakpoints in arcs when moving connecting places or transitions. This can lead to overlapping of arcs. More research into alternative arc pathing could potentially fix these issues.

Figure 7.2: AllVotes after restart of application

7.3.3 Layout

To improve the user experience, the layout below the modeling canvas can be updated to provide a more organized and user-friendly interface. This can be achieved by designing a proper CSS layout that clearly explains the purpose of each field and button, reducing any perceived clutter.

Additionally, updating the default color scheme of the modeling canvas can contribute to a more pleasant and calming environment for users. By selecting a visually appealing color palette, users can concentrate better on their modeling tasks, resulting in increased productivity and overall satisfaction within the application.

7.3.4 Substitution transitions

To enhance the usability of substitution transitions, it would be beneficial to provide a clear visual indicator that distinguishes them from other types of transitions. Additionally, improving the implementation of submodules and their connection to substitution transitions can be advantageous. For instance, when creating a substitution transition, an empty submodule could be automatically generated. The user can then open and start modeling within this submodule. Once the submodule is complete, the user can save it and integrate it into the current working module. This approach streamlines the modeling process

and provides a more intuitive way of handling submodules within substitution transitions.

7.3.5 Saving and loading

Currently, the saving and loading functionality in the application involves synchronizing the model and the JSON representation. To save a file, the user needs to ensure that the JSON representation is up to date with the model and then initiate the download process. Subsequent saves require the user to redownload the file and, if desired, overwrite the initial download.

Implementing automatic synchronization between JSON representation and model and some functionality for saving directly to a user determined file would help with these problems.

7.4 Usability Evaluation

In this section, we explore what we believe the user adoption process for our CPN editor would look like and discuss how new users are likely to approach its various aspects.

During the development process, one of our main focuses was to create a modeling tool that is easy to use for both experienced and inexperienced users in the field of modeling Coloured Petri nets. To achieve this, we implemented drag-and-drop functionality, which is a common and intuitive feature in many modeling tools. We made sure that the places and transitions, which are the most used objects, are easily accessible for the users to add to the modeling canvas. Additionally, we followed standard practices for connecting nodes with arcs, providing clear and easily identifiable ports on places and transitions.

Although the application has not undergone user testing yet, we believe that by incorporating these common modeling practices, the tool is straightforward for users to learn and use effectively.

Regarding the saving and loading functionality, we adopted a slightly unconventional approach approach. Both saving and loading are done locally within the HTML textarea, represents the model JSON format. While this approach may require some practice to become familiar with, it offers a straightforward and intuitive solution for managing the model data.

Saving the model by downloading it as a file is a common practice found in many applications. Similarly, loading a single file is a familiar feature to users. The only difference in our application is that these options are the primary methods for storing and loading models to and from the local system.

When loading a directory that contains a CPN model, the application automatically generates a tree view that represents the hierarchical structure of the model. This provides users with a visual representation of the model's organization. To load specific modules, users can simply right-click on the desired file in the tree view, a practice commonly used in applications for opening files.

Acronyms

CPN Coloured Petri Net.

GLSP Graphical Langugage Server Platform.

PNML Petri Net Markup Language.

Appendix A

Source code and Installation

The source code for the application at the following URL: `https://github.com/smartoceanplatform/cpn-editor`. The repository is currently private, but can be made available if you contact my supervisor.

A.1 Installation guide

Make sure you have Node.js and npm installed. Node.js is available from https://nodejs.org/en.
npm can be installed using your commandline with the following command:
`npm install -g npm`

When you have installed both Node.js and npm it is advised to check your versions and make sure the installations where successful. To do this, use the following commands:

```
node -v
npm -v
```

Per April 2023, the following versions where recommended; node v16.14.2 and npm 8.7.0

Clone the project from github using the link above. Navigate to the electronTrial folder and run the command:
`npm run start`

List of Figures

List of Tables

Bibliography

[1] Étienne André et al. "CosyVerif: An Open Source Extensible Verification Environment." In: *2013 18th International Conference on Engineering of Complex Computer Systems*. 2013, pp. 33–36. DOI: 10.1109/ICECCS.2013.15.

[2] Juan Boubeta-Puig et al. "MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored petri nets." In: *Information Systems* 81 (2019), pp. 267–289. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2017.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0306437917300108.

[3] Alan W Brown and Kurt C Wallnau. "A framework for evaluating software technology." In: *IEEE Software* 14.5 (1997), pp. 37–46.

[4] *Connecting a Smart Ocean*. https://www.investinbergen.no/news-and-events/connecting-a-smart-ocean/. Accessed: 19.05.2023.

[5] World Wide Web Consortium. *Same Origin Policy*. https://www.w3.org/Security/wiki/Same_Origin_Policy. [Online; accessed April 28, 2023].

[6] *CoSyVerif Website*. Website. Accessed: 23.05.2023. URL: https://www.cosyverif.org/.

[7] CPN IDE. *CPN IDE*. https://cpnide.org/. Accessed: 23.05.2023.

[8] *Eclipse Modeling Framework (EMF)*. https://www.eclipse.org/modeling/emf/. Accessed on 2023-05-05.

[9] *ePNK Website*. Website. Accessed: 23.05.2023. URL: http://www.imm.dtu.dk/~ekki/projects/ePNK/index.shtml.

[10] OpenJS foundation. *ElectronJS webpage*. Accessed 11.04.2023. URL: https://www.electronjs.org/.

[11] G Franceschinis, S Donatelli, and R Gaeta. "GreatSPN 2.0: Graphical editor and analyzer for timed and stochastic Petri Nets." In: *Petri Nets 2000* (2000), p. 43.

[12] Andreas Garvik. "A Compiler and Runtime Environment for Execution of Coloured Petri Net Models." In: (2022).

[13] *Graphical Language Server Platform for next-generation diagrams editors*. Accessed: 2023-02-17. URL: https://www.eclipse.org/glsp/.

[14] Dalton Serey Guerrero, Jorge C.A. de Figueiredo, and Angelo Perkusich. "An Object-Based Modular CPN Approach: Its Application to the Specification of a Cooperative Editing Environment." In: *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*. Ed. by Gul A. Agha, Fiorella De Cindio, and Grzegorz Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 338–354. ISBN: 978-3-540-45397-0.

DOI: 10.1007/3-540-45397-0_12. URL: https://doi.org/10.1007/3-540-45397-0_12.

[15] R Heldal et al. "Towards a Formal and Executable Software Architecture Specification of the Smart Ocean Data Service Platform." In: 2023.

[16] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets*. 1st ed. Springer Berlin, Heidelberg, 2009, pp. 1–384. ISBN: 9783642002847.

[17] Kurt Jensen et al. *A tool for editing, simulating, and analyzing Colored Petri nets*. Accessed: 2023-02-17. URL: http://cpntools.org/.

[18] Keila Lima et al. "Marine Data Sharing: Challenges, Technology Drivers and Quality Attributes." In: *Product-Focused Software Process Improvement-23rd International Conference, PROFES 2022*. Vol. 13709. Lecture Notes in Computer Science. Springer, 2022, pp. 124–140. DOI: 10.1007/978-3-031-21388-5_9. URL: https://doi.org/10.1007/978-3-031-21388-5_9.

[19] *npm*. https://www.npmjs.com/. Accessed on May 16, 2023.

[20] James L Peterson and Wolfgang Reisig. "Petri net." In: *Scholarpedia* 2.9 (2007), p. 6474.

[21] Eric Newcomer Philip A. Bernstein. *Principles of Transaction Processing*. 2nd ed. 2009. ISBN: 978-1-55860-623-4.

[22] *Renew - The Reference Net Workshop*. Website. Accessed 23.05.2023. URL: http://www.renew.de/.

[23] *SFI Smart Ocean*. https://sfismartocean.no/. Accessed: May 2, 2023.

[24] Kent Inge Fagerland Simonsen. *elm-pn-editor GitLab Repository*. https://gitlab.com/kentis/elm-pn-editor. Accessed on 2023-05-05.

[25] NORTHWOODS Software. *GoJS - Palette*. https://gojs.net/latest/intro/palette.html. [Online; accessed April 28, 2023].

[26] Northwoods Software. *GoJS A Web FrameWork for Rapidly Building Interactive Diagrams*. Accessed: 2023-02-17. URL: https://gojs.net/latest/.

[27] Spring. *Spring Boot*. https://spring.io/projects/spring-boot. Accessed: May 27, 2023.

[28] *Standard ML Family GitHub Project*. Accessed: 17.04.23. URL: https://smlfamily.github.io/.

[29] *The BETA Programming Language*. Accessed: 17.04.23. URL: https://beta.cs.au.dk/.

[30] Michael Westergaard and Lars Michael Kristensen. "The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator." In: *Applications and Theory of Petri Nets*. Ed. by Giuliana Franceschinis and Karsten Wolf. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 313–322. ISBN: 978-3-642-02424-5.