University of Amsterdam

Formal Verification of components through Mirroring of Coloured Petri Nets

by
Merrick Oost-Rosengren

Academic Supervisor: **Benny Akesson (k.b.akesson@uva.nl)**
Second reader: **Ana Oprescu (a.m.oprescu@uva.nl)**

Host organisation: **ESI (TNO) (https://esi.nl/)**

A thesis
submitted in partial fulfilment of the requirements
for the degree of Master of Science
Software engineering

Amsterdam, Netherlands
July 2024

Abstract

In environments where multiple software or hardware components communicate data with each other (a component-based system), and a breakdown of the components or communication can be fatal (e.g. Military Ships or an Intensive Care Unit), verification of the component model as early in the development process is vital. These components may be developed by different manufacturers or teams, or at different times, so access to other components or the specifications thereof may be impossible. This necessitates a way to validate the component model by itself and confirm that it does not cause deadlocks, or otherwise hinder or is hindered by the functionality of other components, or with/by the data being transferred between components. There seems to be a research gap related to the early validation of component models when data coming from one component affects the behaviour of another component.

This thesis addresses this gap by proposing a novel approach for generating a Verification Model based on the Model of the component. When the two models are combined, we can detect if they run without terminating unexpectedly (are Weakly Terminating) and do not create an infinite amount of execution paths with different data (have a finite State Space).

We introduce two additional types of Coloured Petri Nets (A model which describes a system with data): The Open Coloured Petri Net, which models a component with connections to other components specifically, and the Mirrorable Coloured Petri Net, which contains additional information about the Component which is needed to generate the Verification Model. We call this generation of a Verification Client, Mirroring. We defined a methodology to apply the theory of Mirroring and test the Methodology.

The final objective is to validate that the Methodology can be automated. We concluded that the theory is correct, and a methodology and automation can be applied with a Mirrorable Coloured Petri Net.

Acknowledgements

I foremost would like to thank Benny for his invaluable guidance. Our discussions about the subject were very insightful. Then I would like to thank Deb, who created a verification model to analyse, which allowed me to identify the ruleset and the theory discussed in this thesis.

I would also like to thank everyone who encouraged and supported me: My husband: Johnny, my friends: Erik, Noah, Armin, Sean, Ferdinand, and Holger, the professors and students at CCI, and the AMdEX team, specifically: Thomas, Chris, Tim and Marten.

Table of Contents

Illustrations

Chapter 1

Introduction

In September 1996 a system failure on the USS Yorktown left this cruiser stranded in port over a weekend [1]. The SmartChip project being tested by this ship, was rushed, with no real prototyping, trying to make all parts work together as they went along. The problem occurred when one component communicated an unexpected zero, and another component tried to divide one of its own values by this zero.

The above historical example shows that systems that contain many different components (Component Based System: CBS [2]) may create errors because of data coming from other components. It is vital that these errors are caught as early in the design and development process as possible. Nowadays, a lot of systems are comprised of multiple components. They can contain a plethora of different hardware and/or software components, and the behaviour of one component must not negatively impact other components. If a server deadlocks or crashes, because a client sends invalid or unexpected data, that is a problem. Systems need to be verified to ensure they behave as expected and don't fail when connected to other systems. But during the early phases of a project (e.g. Inception, Analysis, Design), there may not be other systems available, they may be created by different manufacturers and only start communicating when integrated by an organisation. For instance, in the Intensive Care unit of a hospital, there are a lot of systems from different manufacturers working together, some to monitor blood oxygen, some to monitor heart rate, and some to display information to medical professionals. Some of these systems may only be created later based on an interface definition.

This brings us to the topic of this thesis: How to verify a single component, which is part of a CBS which communicates data with other components, before building it, and when connecting components are not available, and ensure that a component does not deadlock or crash, specifically in environments where this could be catastrophic.
One of the problems of analysing CBS where data is being communicated, is that data can have many different values. This makes it useful to abstract the data. That communication does not look at every value of the data being communicated but refers to this as '*name of datatype*' and all data of this type is being treated equally. The problem of different data is being delegated to a document describing the datatype (the interface definition), but the system is not checked if it handles all values of the datatype correctly until implementation (using unit tests), if it is checked at all. A component may need to behave differently depending on what data is being received, so ignoring the contents of a data element may not be desired.

In this thesis, we will investigate a novel approach to verification by modelling the Interface of a system or the whole system as a Coloured Petri Net (CPN), then check this CPN for weak termination. A CPN can be seen as a model that has both a graphical and a scientific notation (see Chapter 2.4). Weak termination of a CPN defines that from each reachable state of the system, a final state can always be reached (see Chapter 2.3.5). Checking for weak termination, of an interface when the accompanying interface of the connecting system is not available during this check, has a problem: A final state can not be reached without communication with the connecting system. So part of this novel approach will be the generation of a connecting Interface or Client, specific for verification, based on the definition of the original Interface or Client. We will call this generation "Mirroring".

Research questions:

1. How can a Component or the Interface of a Component, which exchanges data with other components in a Component-based system, and this data changes the behaviour of this component, be verified for weak termination?
2. Can rules be identified to generate a component model from the original component model which, when fused with this component model can verify for weak termination?
3. Are these rules strict enough to be implemented as an algorithm? How would a program applying this algorithm be defined?

Research contributions:

1. Definition of an Open Coloured Petri Net (OCPN) that extends the CPN definition with Input and Output Places. Definition of a Mirrorable Open Coloured Petri Net (MOCPN) that extends the OCPN with the archetypes of data movement in a system, which are needed during mirroring.
2. The concept of Mirroring a MOCPN without a client, to allow verification of a Component Model in a CBS, using weak termination. By generating a (Mirrored) verification MOCPN
3. The process and rules of Mirroring and The interactions of Guards and Arc Inscriptions during Mirroring and creating a Prototype to validate this process and these rules.

The thesis is structured as follows:

- In **Chapter 2** we discuss the preliminary concepts on which this Thesis is build.
- In **Chapter 3** we will introduce Mirroring and Mirroring definitions.
- In **Chapter 4** we define the methodology to apply Mirroring for Verification Client generation, and a software tool which applies this theory.
- In **Chapter 5** we will use this tool to investigate the validity of the theory, and identify some of the limits.
- In **Chapter 6** we look at related work.
- In **Chapter 7** we will answer the research questions, including further theorising the usefulness and limits of Mirroring for Verification Client generation. We will also identify some subjects which we could not address in this Thesis.

Chapter 2

Preliminaries

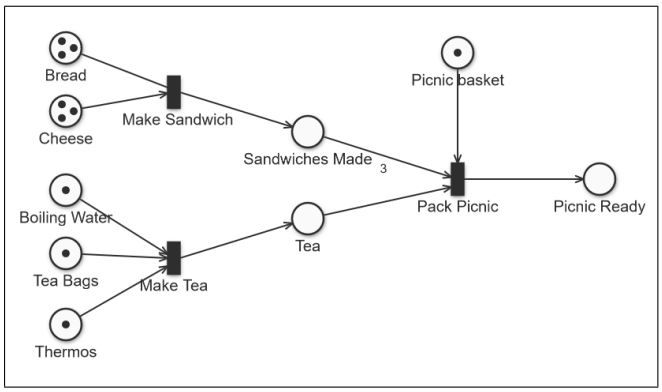

Figure 2.1: Picnic

In this chapter, we will explain the existing theories of Petri Nets, Coloured Petri Nets, and Inversion of Coloured Petri Nets, which are the basis for understanding the theory of mirroring to create a verification client.

Example: During the organizing of a picnic, there are tasks involved, some of which can only start after others have been completed. Others can be done at any time. A way to visualize this is shown in Figure 2.1. It has circles where things are available, and squares where activities for the picnic are done.

2.1 Component-Based Development

Component-based development (CBD) is a software engineering approach that emphasizes the reuse of software components to build larger software systems. In component-based development, a software system is decomposed into smaller, modular components that can be developed, tested, and maintained independently of each other. These components can be combined and configured to build a larger system, reducing the complexity and effort involved in software development. Different formal models supporting component-based development have been proposed, like Cadena [3] and SaveCCM [4] This thesis specifically looks at systems that consist of two components: the client (a component that needs a service) and the server (a component which supplies a service), and how to validate the proper function of one of these when the other does not exist, by calculating/generating a version of the other component specific for verification.
In the picnic example 2.1 the activities can be seen as different components, a different person can be responsible for making the sandwiches, making the tea and packing the picnic. These people are working together to make the picnic, each with their own activity.

2.1.1 Asynchronous communication systems

Asynchronous communicating systems are a type of system in which components or processes communicate with each other through asynchronous message passing. In an asynchronous communication system, processes can send and receive messages at any time, independent of each other, and without the need for global synchronization. Asynchronous communicating systems are commonly used in distributed systems, concurrent programming, and software engineering, where components or processes need to interact with each other.
In the picnic example 2.1 it does not matter if the tea is made before during or after the sandwiches are made, only that everything is made before the picnic is packed.

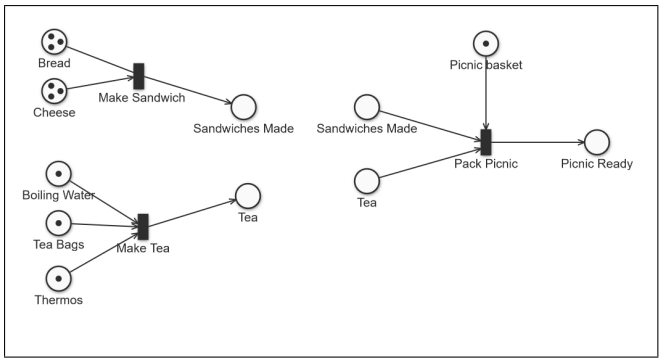

Figure 2.2: Picnic Components

2.1.2 Software interfaces

a software interface is a defined boundary between two or more software components or systems, through which they can interact with each other. The purpose of a software interface is to provide a standard way for components or systems to communicate and exchange information, regardless of the underlying implementation details. In asynchronous communication systems, it is the definition of the messages being sent. It is also the name of the location (Place in a Petri Net, see below) which sends a message or receives a message.
When redrawing the picnic example as Figure 2.2, "Sandwiches Made" and "Tea" can be viewed as the boundary of the "Make Sandwich" and "Make Tea" systems. The person making the tea places it in "Tea" and the person who packs the picnic takes it from "Tea".

2.2 State Machine

A state machine is a mathematical model used to represent the behaviour of a system. It defines a set of states, which represent the possible configurations of the system, and a set of transitions, which represent the possible changes between the states.

In a state machine, the system can be in only one state at any given time, and transitions can occur only between adjacent states. The transitions can be triggered by events or conditions and can cause actions or side effects to occur in the system.

The picnic example 2.1 is a state machine. The different states of a State Machine can be visualized with a State Space Graph, which is described later in the Petri Nets chapter 2.3.4.

2.3 Petri Nets

The picnic example 2.1 is a state machine, but to be more precise, it is a Petri Net. In computer science, engineering, biology, and others this is used to understand and manage complex processes. In this system the checklist items are represented with circles called "places", these places hold check marks called "tokens". There may be more tokens per place, for instance, the place "sandwiches made" can contain one token per sandwich made, like a counter. Then some activities have to be done, which are represented with squares called "transitions", for instance, "make sandwich", or "pack picnic" which should only be done once the predetermined number of sandwiches and drinks have been made. Finally, there are lines connecting the places to the transitions called "arcs", which guide the process of making the picnic. So this Petri Net is an interactive map of tasks that need to be done to create a great picnic.

A Petri Net, is a Mathematical and Graphical notation to describe a state machine, it consists of data (called tokens), Places that can contain tokens, Transitions that change the state of the machine, and the relation between Places and Transitions (called Arcs). A Petri Net starts with a number of tokens in places, which defines the starting state. During execution, one Transition that is enabled (based on the state) is Triggered, which moves and/or changes tokens to other (or the same) Places. After the Triggering of a Transition, the Petri Net has a new State. This cycle of Triggering transitions and moving tokens continues until a pre-defined end state is reached, or no further Transitions are enabled.

Carl Adam Petri, who gave his name to the Petri Net, first formally introduced this type of net in his paper "Kommunikation mit automaten" [5][6]. The notation gets more formalised in his later work [7]. Other early works on the theory of Petri Nets come from Wolfgang Reisig [8][9].
The original Petri Net is focused on token communication and manipulation without a value associated with the token. This Thesis focuses specifically on Coloured Petri Nets, which replaces tokens with coloured tokens where the colour specifies the data. A lot of theory on Coloured Petri Nets is written by Kurt Jensen and collaborators [10].

2.3.1 Terms used describing Petri Nets

Tokens

A token is a symbol that represents the presence or absence of a resource or token type within a place. Tokens are placed in the places and flow through the Petri Net (or are created/destroyed) during Execution.

Tuple

A tuple when used in a Coloured Petri Net is a Token that contains multiple values, which can be used to combine multiple tokens into a single token. This is particularly useful when multiple tokens are transferred between two Coloured Petri Nets.

Places

A Place can have 0 or more tokens defined for its Initial State. During the execution of the Petri Net, this usually changes. In programming, a Place can be seen as a variable, where the colour defines the type, and the token defines the value.

Arcs

Arcs transport tokens to and from Transitions. An Arc is always from a Place to a Transition or from a Transition to a Place. An Arc is never from a Place to a Place or from a Transition to a Transition. An Arc has an inscription which can contain rules, and the name by which the Transition refers to the Token transported

Transitions

Transitions define when Tokens move between Places. Tokens can only move if all the Incoming Arcs of the Transition have a Token in the connecting Place. Additionally, a Transition can have a Guard, which is a Boolean function on the incoming Tokens, and only if this Guard evaluates to True the Transition is said to be Enabled and Tokens can move across the Transition. The inscription on the outgoing Arcs then defines the Tokens that move to their connecting Place.

Transition Trigger

During execution only the movement of tokens across one Transition is calculated at a time: this Transition is Triggered. After a Transition is triggered, the new state of the Petri Net is calculated, and based on this, a new Transition is selected to be Triggered. There is not an assigned order in which Transitions are Triggered; any enabled Transition can be Triggered during execution.

Nodes

Node is a generic term for Places and Transitions.

Colours

For Coloured Petri Nets, defined later in this chapter, a token represents a specific data value. This value is said to be its Colour, and every Colour is part of a Colourset (data type). A (classic) Petri Net can be defined as a Coloured Petri Net with one Colourset which contains one Colour: Black.

2.3.2 Petri Net

Scientifically a Petri Net is a tuple N with the following definition. N = net structure. $N = (P, T, F)$
P = set of places
T = set of transitions
F = set of arcs. $F \subseteq (P \times T) \cup (T \times P)$

$P \cap T = \varnothing$
node $\in (P \cup T)$
${}^\bullet x$ = pre-set of a node. ${}^\bullet x =_{\text{def}} \{y | (y, x) \in F\}$ [8]
$x^\bullet$ = post-set of a node. $x^\bullet =_{\text{def}} \{y | (x, y) \in F\}$ [8]

2.3.3 Open Petri net

In our second picnic example (figure 2.2), the person is not making it alone, but they have their kids helping them. One is making the sandwiches, one is making the drinks, and one is putting everything in the basket. The activity of packing the basket depends on the other activities. In this case, the place "Sandwiches Made" of the "Make Sandwich" Petri Net is linked to the place "Sandwiches Made" of the "Pack Picnic" Petri Net. The place is called "Output Place" in the "Make Sandwich" Petri Net, and "Input Place" in the "Pack Picnic" Petri Net.

An open Petri net (OPN)[11] is a subclass of Petri nets which is suitable for modelling client/server systems. An OPN is a tuple N with the following definition:
$N = (P, I, O, T, F)$
P = set of Internal places
I = set of Input places with ${}^\bullet I = \varnothing$
O = set of Output places with $O^\bullet = \varnothing$
T = set of transitions
F = set of arcs. $F \subseteq ((P \cup I \cup O) \times T) \cup (T \times (P \cup I \cup O))$
$I \cup O$ = the interface places of the net
The sets P, I, O and T are pairwise disjoint.

note: $P \cup I \cup O$ is the set of all places, which is defined as P in the classical definition of a Petri Net (chapter: 2.3)

Software Interface of an Open Petri Net

The software interface of an OPN (chapter: 2.3.3) includes a set of input and output places, which serve as the boundary between the OPN and the external software components. The input places receive input data from the external software components, while the output places send output data back to the external software components.

Two OPNs can be composed by fusing their shared interface places. We say two OPNs are composable if and only if the set of input places of one net is equal to the set of output places of the other net, and vice versa.[11]

The input and output places of an OPN are typically connected to the transitions of the OPN, which perform the actual processing of the input data and produce the output data. The connections between the input/output places and the transitions of the OPN form the interface specification, which defines the behaviour of the OPN from the perspective of external software components.

The interface specification can be designed to be flexible and customizable, allowing different external software components to interact with the OPN using different protocols or formats. The interface can also be designed to be scalable and extensible, allowing additional input/output places and transitions to be added to the OPN as needed to support new functionality or integration with other systems.

Fusing two Open Petri Nets

Fusing two Open Petri Nets combines them into one Petri Net. Fusing combines the Output Places of one net with the Input Place into one Place. This Thesis only looks at fusing nets with a 1 to 1 mapping between Output places and Input Places.

If there are two (fusible) nets:
$N_1 = (P_1, I_1, O_1, T_1, F_1)$
$N_2 = (P_2, I_2, O_2, T_2, F_2)$
Then:
$N_{fused} = (P_1 \cup P_2, I_1 = O_2, O_1 = I_2, T_1 \cup T_2, F_1 \cup F_2)$

2.3.4 State Space

The state space[10] of a Petri Net lists all the states a Petri net can be in. A state in a Petri net is determined by the tokens in different places. The size of the Statespace is determined by the number of places in the Petri Net and the number of tokens of each colour that can be present in each place. The state space of a Petri Net can be represented using a state space graph or reachability graph, which shows the possible transitions between markings of the Petri Net. Figure 2.4 and 2.5 show two states of a coloured Petri Net which we will explain in the next section.

Reachability Graph and State Space Graph

For a state machine, a state space graph is typically represented as a directed graph, where the nodes (circles/squares) represent the states, and the edges (arrows) represent the transitions between the states. Each node in the graph represents a single state, and each edge represents a possible transition from one state to another.

A State Space Graph can be created based on an initial state, and only contain all states which are possible based on that initial state. But a State Space Graph can also be created based on all possible states of the State Machine. Depending on the state machine, this Graph can contain a huge amount of nodes and edges.

A Reachability[10] Graph is another name for a state space graph and shows the possible transitions between markings of a Petri Net. In a reachability graph, each node represents a unique marking of the

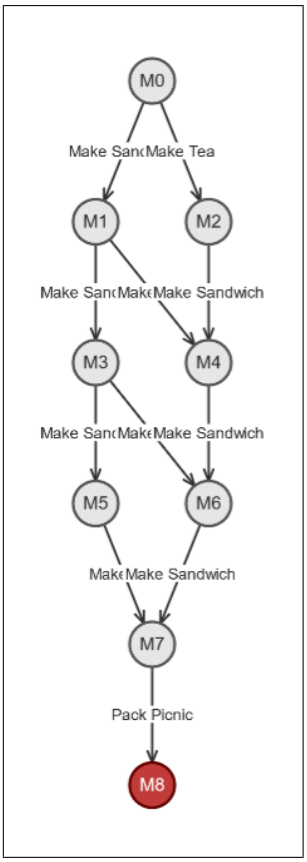

Figure 2.3: Picnic

Petri Net, and each edge represents a possible transition from one marking to another. The reachability graph can be used to analyze the behaviour of the Petri Net and to identify properties such as deadlocks and weak termination.

Figure 2.3 shows the state space/reachability graph for the picnic example.

2.3.5 Weak Termination

Weak Termination[11] is the property that from any reachable state in a Petri Net it will always be possible to reach a final state of the Petri Net. This is opposed to Strong Termination which requires that the Petri Net reaches a terminal marking from any initial marking.

This Thesis only regards Closed Nets, meaning that the Final State is the same as the Start State. This will allow the Petri Net to run forever without generating a Deadlock or a State Space Explosion. (Defined below)

For Non Closed Nets, Weak termination guarantees that the system will eventually come to a stable state from which no further transitions are possible.

Deadlock

A Deadlock[12] occurs when all the tokens in the Petri Net are blocked and unable to move to any other places, and hence no Transition is enabled. Note: Reaching a terminal marking and terminating is desired, but a deadlock occurring at any other time is not.

State Space Explosion

A State Space Explosion occurs when the Petri Net generates new tokens and states without reaching a Final State. We also use the term State Space Explosion when the number of states being generated is so numerous that analysis (even with a computer) will take an excessive amount of time.

2.4 Coloured Petri Net

In the picnic example, sandwiches and tea can be regarded as picnic items and more picnic items can be envisioned, like a picnic blanket. Rewriting the left side with that in mind creates figure 2.4. Note: The green elements are discussed later. In this diagram, every Place has more information. The place ”Bread” has a marking ”3‘()” and a marking of ”BREAD”, which means there are three tokens of type ”BREAD”. ”BREAD” is called the Colour Set. For ”BREAD” this works the same as in the original Petri Net, there is only one type of ”BREAD”, so the ”BREAD” tokens are said to have the colour black. Black tokens in a CPN are identical to tokens in a Petri Net. The colour set ”PICNIC_ITEMS” can have tokens of colour ”sandwich” or ”tea”, which can’t be seen in this figure, so let’s show them. Figure 2.4 contains green elements, these are used for simulating the CPN, and these are the start tokens of the simulation and this is called the start state. Executing this simulation to its final state creates figure 2.5. In the final state, we see ”(4) 1‘tea ++ 3‘sandwich” which means that ”picnic items” now contain 4 elements, of which 1 has the colour tea and 3 have the colour sandwich.

A Coloured Petri Net[10] is a 8-tuple $\mathcal{N} = (P, T, A, \Sigma, V, C, G, E)$, where:

1. P is a finite set of places.

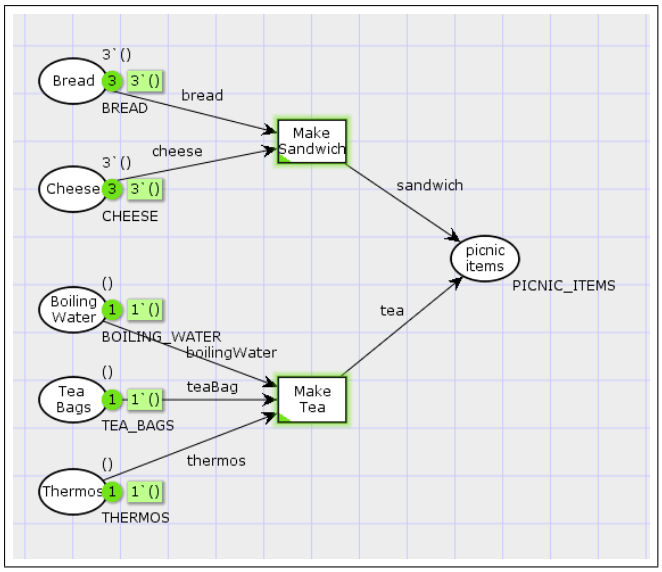

Figure 2.4: Picnic CPN start state

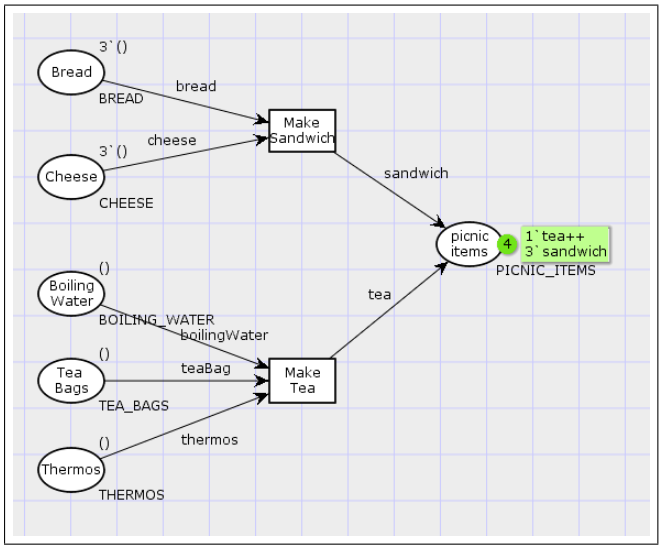

Figure 2.5: Picnic CPN end state

2. T is a finite set of transitions T such that $P \cap T = \emptyset$.

3. $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs.

4. Σ is a finite set of non-empty colour sets.

5. V is a finite set of typed variables such that Type $[v] \in \Sigma$ for all variables $v \in V$.

6. $C : P \to \Sigma$ is a colour set function that assigns a colour set to each place.

7. $G : T \to EXPR_V$ is a guard function that assigns a guard to each transition $t \in T$ such that Type $[G(t)] =$ Bool

8. $E : A \to EXPR_V$ is an arc expression function that assigns an arc expression to each arc $a \in A$ such that Type $[E(a)] = C(p)$, where $p \in P$ is the place connected to the arc a

When a transition $t \in T$ is enabled in a CPN, it can be fired, which means that the tokens in its input places ($p_{in} \in P$ where there exists an arc $(p_{in}, t) \in A$) are consumed, and tokens are produced in its output places ($p_{out} \in P$ where there exists an arc $(t, p_{out}) \in A$)), according to the following rules:

- The tokens in the input places must match the specified colour and value constraints of the transition's input arcs. If any token in the input places does not match the constraints, the transition cannot fire.
- If the input tokens match the constraints, the transition can fire, and the input tokens are consumed. The transition then produces output tokens in the output places, according to the constraints of the output arcs.
- If multiple transitions are enabled at the same time, only one of them can fire, and the choice of which transition to fire is determined by a scheduling policy, such as random selection or priority.

2.4.1 Unrolling a CPN into a Petri Net

Unrolling a CPN into a Petri Net involves transforming the CPN into an equivalent Petri Net that preserves its behaviour and properties but does not use colours to differentiate tokens. This allows a CPN to be analysed with the techniques and tools designed for Petri Nets. The unrolling process converts each coloured place and transition of a CPN into multiple places and transitions in a Petri Net, one for each colour present in the original CPN. "MCC: A Tool for Unfolding Colored Petri Nets in PNML Format" [13] explains the unrolling process in detail.

2.4.2 Inverse of a Coloured Petri Net

The inverse of a CPN is simply put the CPN running backwards. The inverse of the previous CPN picnic example 2.5 is shown in figure 2.6. All the arrows point in the opposite direction, and the start state

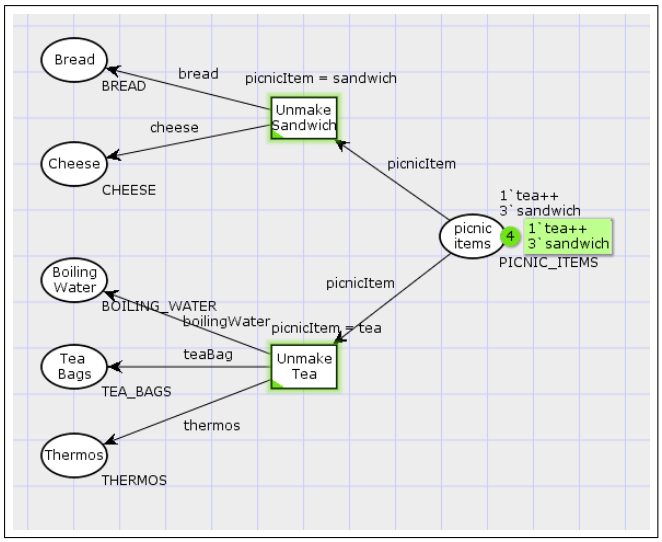

Figure 2.6: Picnic CPN inverse start state

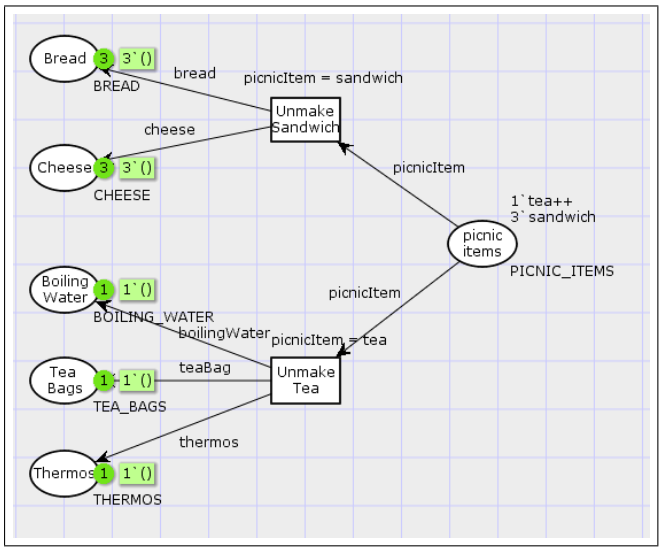

Figure 2.7: Picnic CPN inverse end state

has the tokens in the "picnic items" place. Executing this CPN results in the final state shown in figure 2.7. So the inverse completely undoes the process which the original CPN did. The "picnicItem = sandwich" is a guard to ensure that the Inverse only unmakes a Sandwich when the picnicItem is a sandwich.

This Thesis only looks at Transition Inversions which can be directly inferred, (called Trivial and Basic Transformation). There are more elaborate ways to calculate inversions, as investigated in the papers of Khalfaoui [14], Bouali, Barger and Schon [15] [16], Bouali, Rocheteau and Barger [17].

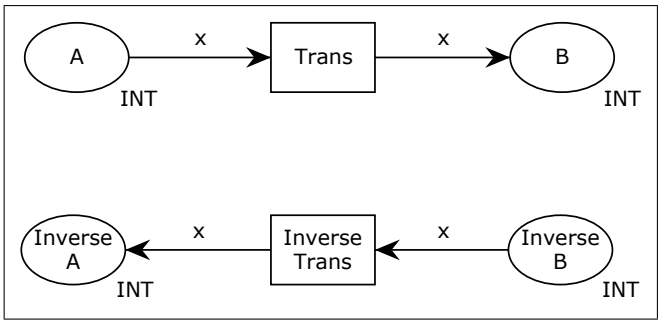

Figure 2.8: Transition with 1 var and Inverse

Trivial and Basic Transitions

A Trivial Transition is when one token is passed through a transition (figure 2.8). The inverse of this is itself with the arcs inverted. If the Transition has a related Guard, this stays the same.

A Basic Transition can have more input and output places. For our research, we are interested in Basic Transitions with outgoing constants and Constants in the Transition Guards. These have an important place when communication between client and server uses one input and one output port.

When a Transition also sends a constant to another place (figure 2.9), we know that the value coming from C is always 10, but we do not know if C can receive any other values, so we will need to Guard against other values. Hence the inverse of this transition is a Transition with a Constant as Guard (figure 2.10)

The opposite is also true, when a Transition has a Constant as Guard (figure 2.10), then we know the value of the associated variable (y). So the inverse of this transition, is shown in figure 2.9, where the variable (y) on the inscription is replaced with the Constant that was part of the Guard.

We can also combine multiple input tokens into one token (a tuple) which is shown in figure 2.11.

Inversion calculation

The inverse (t') of Transition t can be calculated as follows:
Let

- a be the set of all the **incoming** token values for t.
- b be the set of all **outgoing** token values for t

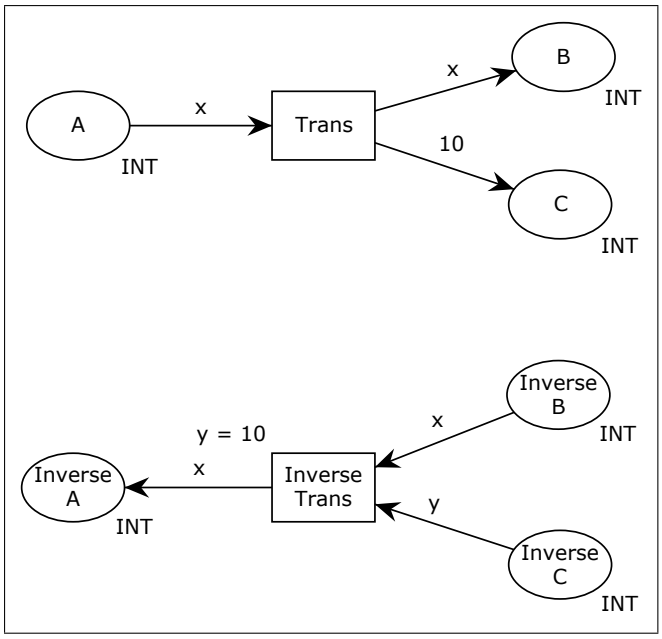

Figure 2.9: Transition with 1 outgoing constant and Inverse

- t be a transition which maps a to b such that $t(a) = b$
- $g(a)$ be the guard function for t
- all values/formulas above are known

and

- a is also the set of all the **outgoing** token values for t'.
- b is also the set of all **incoming** token values for t'.
- t' is the mirror transition which maps b to a such that such that $t'(b) = a$
- $g'(b)$ is the guard function for t'

then

- Solve t' where $t'(b) = a$. This should give the outgoing arc inscriptions.
- Solve g' where $g'(b) = g(a) = g(t'(b))$

Example: We have a transition which is triggered when the input token (x) is larger than 10, this then calculates an output token(y) which is 2 times the input token.

- $a = x$
- $g(a) : x > 10$

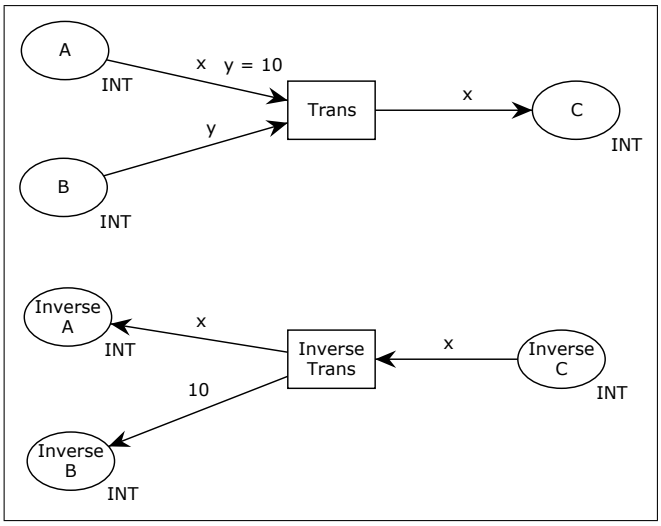

Figure 2.10: Transition with a Constant as Guard and Inverse

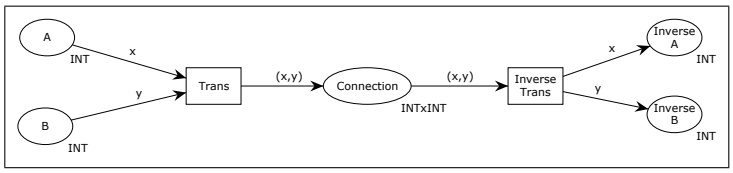

Figure 2.11: Using a tuple to combine tokens

- $b = y$
- $t(a) : y = 2x$

Inverse:

- Solve t' by substitution using $t(a)$, b, and a.
  - $t'(b) = a$
  - $t'(b) = x$
  - $t'(b) = \frac{y}{2}$
- Solve g' by substitution using $g(a)$ and $t'(b)$.
  - $g'(b) = g(a) = g(t'(b)) : x > 10$
  - $g(\frac{y}{2}) : x > 10$
  - $g'(b) : y > 20$

Example: We have two input places with tokens x and y and we want to combine these in a tuple. In the mirror, we will receive the tuple and extract the two tokens again (figure 2.11)

- $a = x, y$
- $g(a) : true$ (no guard)
- $b = z$

- $t(a) : z = (x, y)$

Inverse:

- $t'(z) : (n_1, n_2)|x = n_1|y = n_2$
- $g'(z) : true$

Which can be written as the original transition with the arcs moving in opposite direction

Chapter 3

Mirroring

We will now discuss the way we can create a Client based on the definition of the CPN of the server (Specifically a Mirrorable Open CPN, which will be defined later in this chapter). We start with the definition of an OCPN. Then we look at an analogy, to get a feeling for the difference between a Mirror and an Inverse, and the relationship between arc annotations and guards. Then we will go into more detail about what we want to accomplish and how, and define the definition of an MOCPN

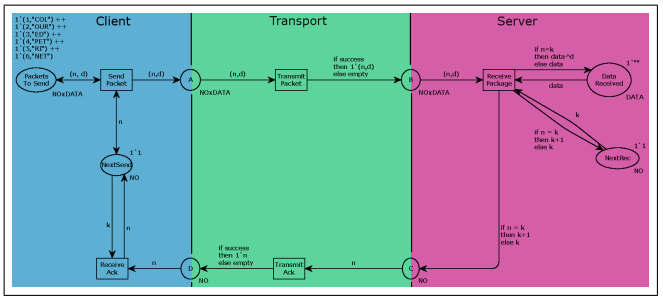

Figure 3.1: Communications Protocol with Systems

3.1 Open Coloured Petri Net (OCPN)

Like an OPN compared to a PN, an OCPN is a CPN with Input and Output Places.
The Communication Protocol Petri Net defined by Jensen [10] models a process of 2 computers (a client and a server) and their communication, as shown in figure 3.1. When we are only modelling one of the systems (e.g. the Client), the resulting Petri Net contains a gap. It has Places A and D which consume and generate tokens, but they don't seem to have a way to do this. However as we can see from the figure, they are places where they connect to another Petri Net, and this is where they send and receive tokens from. We are calling this type of Petri Net an Open Coloured Petrinet and A and D Input and Output Places. So the figure shows 3 Open Coloured Petri Nets and the interfaces between them.

The definition of a OCPN: $\mathcal{N} = (P, T, A, \Sigma, V, C, G, E, I, O)$, where:

1. P is a finite set of places.
2. T is a finite set of transitions.
3. A are all Arcs
4. Σ is a finite set of non-empty colour sets.
5. V is a finite set of typed variables such that Type $[v] \in \Sigma$ for all variables $v \in V$.
6. $C : P \to \Sigma$ is a colour set function that assigns a colour set to each place.
7. $G : T \to EXPR_V$ is a guard function that assigns a guard to each transition $t \in T$ such that Type $[G(t)] =$ Bool
8. $E : A \to EXPR_V$ is an arc expression function that assigns an arc expression to each arc $a \in A$ such that Type $[E(a)] = C(p)$, where $p \in P$ is the place connected to the arc a
9. $I : I \subseteq P$ is the set of Input Places.
10. $O : O \subseteq P$ is the set of Output Places.

Such that:

11. $P \cap T = \emptyset$.
12. $I \cap O = \emptyset$.

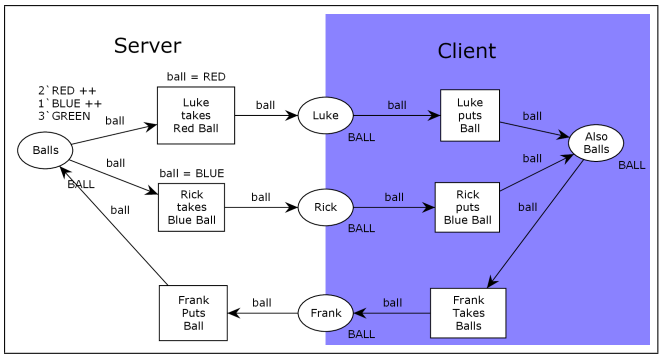

Figure 3.2: Moving Balls Analogy

3.2 Analogy of Mirroring

As an analogy, we are going to use moving a ball between two containers by three persons (figure 3.2). We have divided the figure in a Server side and a Client side. We see that the Server and the Client side are both OCPNs where Luke/Rick/Frank are Input and Output places. If we look at the Server and ignore the Client, we find that Luke and Rick have kept taking balls, and Frank does not have balls to put back. We add a place for Luke and Rick to put the balls "Also Balls" and we have Frank take balls from this place. Now we have a way for the balls to move around without restriction.

When looking at the client, we see that it is the Inverse of the server. The "... Puts Ball" and the "... Takes Ball" transitions are each other's inverse/mirror transitions. "Luke", "Rick" and "Frank" are the Input and Output places of the Server and Client. When we simulate this Petri Net, the balls can move freely, so the red balls can be in "Balls", "Luke", "Frank" or "Also Balls" place (or any combination), and the blue ball can be in "Balls", "Rick", "Frank" or "Also Balls" place (or any combination), the green balls stay in "Balls" and won't move. Because the balls can move freely, and without much control, we regard this as an asynchronous system. This may be desired, but most of the time, when creating a client/server system, we want more control, the server sends data (a ball), and the client responds with (a ball), we regard this as a synchronous system. To create a synchronous system, we have to add more control, as shown in figure 3.3

In figure 3.3 additional places and arcs are included, which we call "Process Flow" since it controls when tokens (data/balls) are sent/received from/to the client. The Mirrored Client also includes a Process Flow which activates the mirrored transitions on the client in the same order as the Process flow activates the transitions on the server, this is the difference with an Inverse Client (figure 3.4). While with the mirrored client the balls move as expected, with only one ball moving to the client and back, in the inverse client, the balls stop moving, because the transitions on the client are triggered in the wrong order. In these figures, the blue-coloured arcs are inversed, while the black-coloured arcs are not inversed.

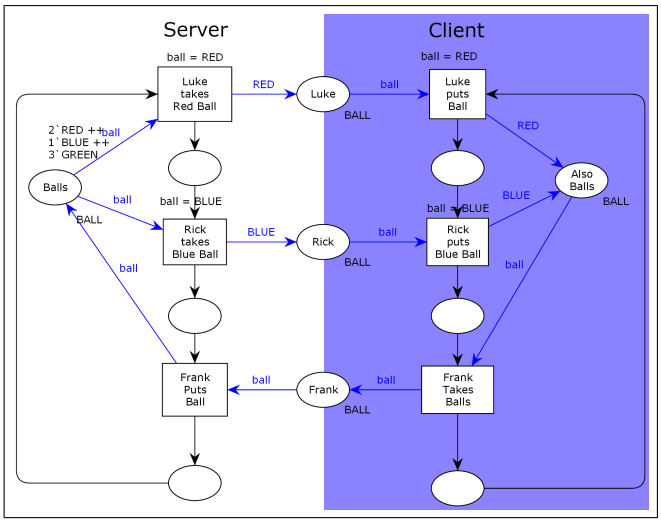

Figure 3.3: Moving Balls Analogy Mirror

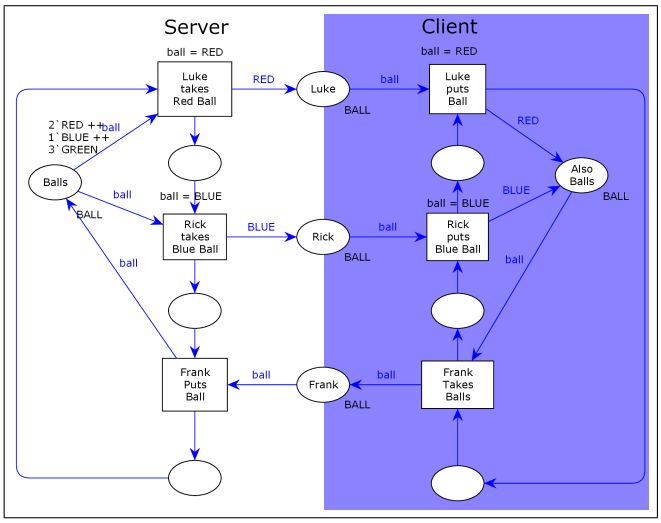

Figure 3.4: Moving Balls Analogy Inverse

The annotations of the arcs have changed from "ball" to "RED" and "BLUE", this is to assist in the (automatic) mirroring process. This information is needed during mirroring. In figure 3.5 the relationship between those changed annotations with the guards of the transitions is shown. The annotation of the arc

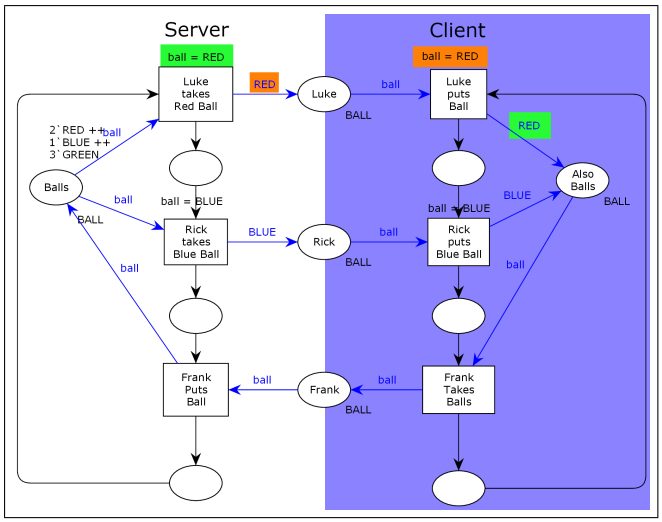

Figure 3.5: Moving Balls Analogy: Guards and Arcs

”RED” on the server gets translated in the mirror to ”ball = RED” (Marked with orange boxes), and the guard of ”ball = RED” on the server gets translated in the mirror to the arc annotation ”RED” (Marked with green boxes).

Side note: The updated annotation of the arcs on the server could be derived from the guards on the server, but this is not always the case. We can envision a pre-processing step which would derive these types of annotations before mirroring. This is out of scope for this thesis, since it specifically focuses on the Mirroring itself.

Finally, in this CPN, the server sends two balls, while the client only sends one ball back. This will eventually lead to the situation that the ”Balls” place will run out of BLUE or RED balls, and either Luke or Rick can no longer take a ball. So this Petri Net does not weakly terminate. This is what the process of creating a Mirrored Client is checking. In this case, we could resolve this by adding a player who also moves a ball back.

3.3 Mirroring Elaboration

When we have a Client/Server system, and create one of its parts, let’s say the Server. To verify the functionality of the server, we also need a client. Depending on the complexity of the Server, a client may not be available. If we have the CPN of the Server, we can create the CPN of the client by using Mirroring. This mirror is suitable for exercising the possible states of the CPN of the Server.

To explain Mirroring, let’s look at what we want to accomplish:

1. Tokens from a place on the server that passes through an Input/Output place should end in place on the Mirrored client.
2. Transitions triggered on the server related to communication should also trigger on the Mirrored client.
   (a) The transitions on both sides should trigger in the same order, to keep the systems synchronised.
3. If tokens are combined in a tuple on the server, these tokens should be extracted on the Mirrored Client.
4. If the Server needs to do a specific action for the client (e.g. do a complex calculation), the Mirrored client should not do the same action. (See below for an example)

In 1 and 3, we can see that these work exactly the same as with Inversion 2.4.2, but 2 works differently from inversion. In an inversed CPN the transitions are triggered in reverse order, and that is not desirable during Mirroring. Thus when creating a mirror, only the flow of tokens that move between the two open nets should be inverted. The significance of that is demonstrated with Figures 3.6 and 3.7. The only difference between the two is the direction of the arcs between CSend, CA, CReseive, CB. In the Mirrored Client, this is the same as in its Server, but in the Inverted Client, this is the inverse of its Server. If both Clients start with a token in CA, then the Inverted Client will deadlock, waiting for an ACK token from the Server. The Mirrored Client however sends the data and receives the ACK as expected. If the Inverted Client would have started with a token in CB then it would be functioning identically to the Mirrored Client. For a simple send/receive loop this is often correct, but as soon as we introduce additional communication steps 3.12, this does not hold.

4 is only useful when creating a Mirrored client for verification. In general, we expect the Server to do an activity for the client. If we mirror all arcs and transitions, the client may be created to also do this activity itself, or it may be created in a way that does not allow proper validation of the Petri Net. e.g. If

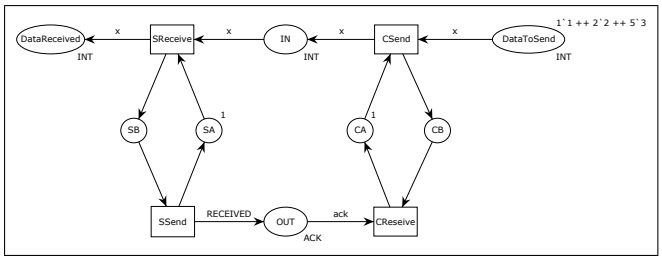

Figure 3.6: Server with Mirrored Client

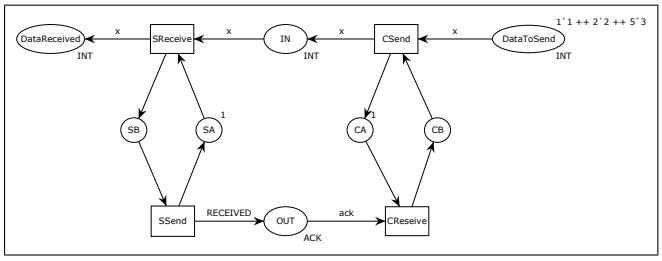

Figure 3.7: Server with Inverted Client

the function of the server is to add 2 numbers and return the result, then the Mirrored client should not also add the 2 numbers and calculate its own result. If we would create a Mirrored client without removing the related arc, the Mirrored client would: Calculate the result itself and/or Tries to separate the result value into the original values. Neither is useful for verification, so the related arcs should not be copied to the Mirrored Client.

During Mirroring, the Input Places of the Original are identical to the Output Places of the Mirror and vice versa. The Client Server Data Flow in a Mirror is inverted while the Internal Data Flow is the same as in the original CPN and the order in which the Interface places are accessed is also the same (controlled by the Process flow). The Client Server Data Flow is defined as Tokens which Flow between two Places in two different Open Coloured Petri Nets to transfer data between those Places. The Process Flow is defined as the order in which transitions handling the Client Server Data flow are triggered, usually defined with black tokens. This is further explained in 3.5.

There is one additional transition that may be useful during mirroring which should not be used during inversion. This is when the data flows through a Transition and sets a data place (3.11). This transition should be handled with suspicion since it does not transfer data between the Petri Net and its Mirror, but it sets the dataplace in the Petri Net and

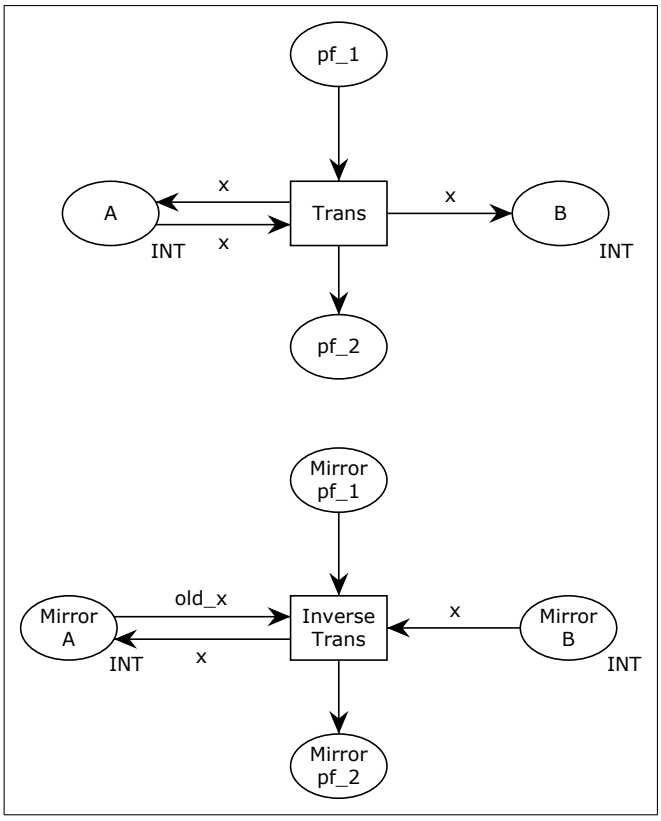

Figure 3.8: Transition with 1 var and Mirror

the Mirror to the same value. This can be useful when the tokens that are being transferred are not inserted in a data place, but for instance in the interface places during verification.

3.4 Definition of a Mirrorable Open Coloured Petri Net

To enable mirroring we have to add additional information to the OCPN tuple defined earlier. This additional information can sometimes be inferred from the Petri Net itself, but not always. In the mirror some arcs stay the same, some are inverted and some are removed. Compared to the definition of an OCPN we split A into multiple parts depending on the function of the Arc. The definition of a MOCPN: $\mathcal{N} = (P, T, A_p, A_d, A_1, \Sigma, V, C, G, E, I, O, M)$, where:

1. P is a finite set of places.
2. T is a finite set of transitions.
3. A_p are all Arcs which are for Process flow
4. A_d are all Arcs which are for Client Server Data

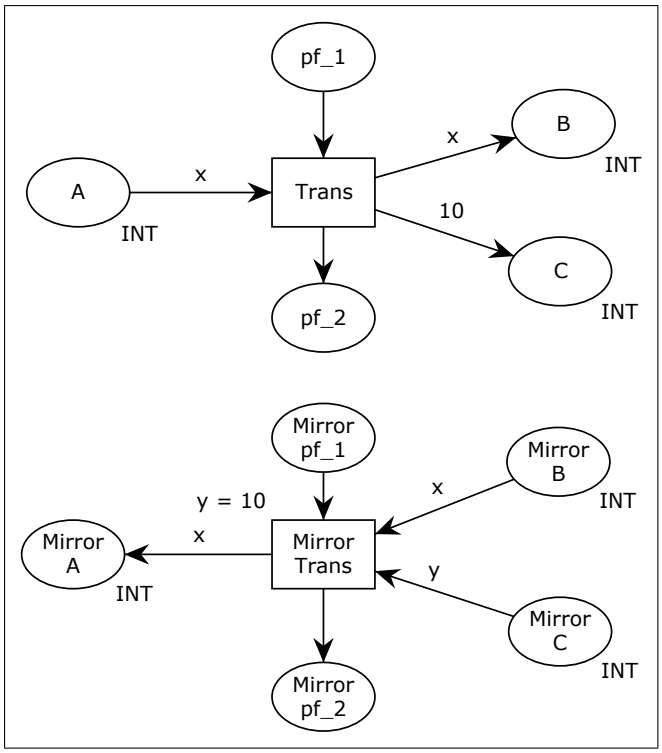

Figure 3.9: Transition with 1 outgoing constant and Mirror

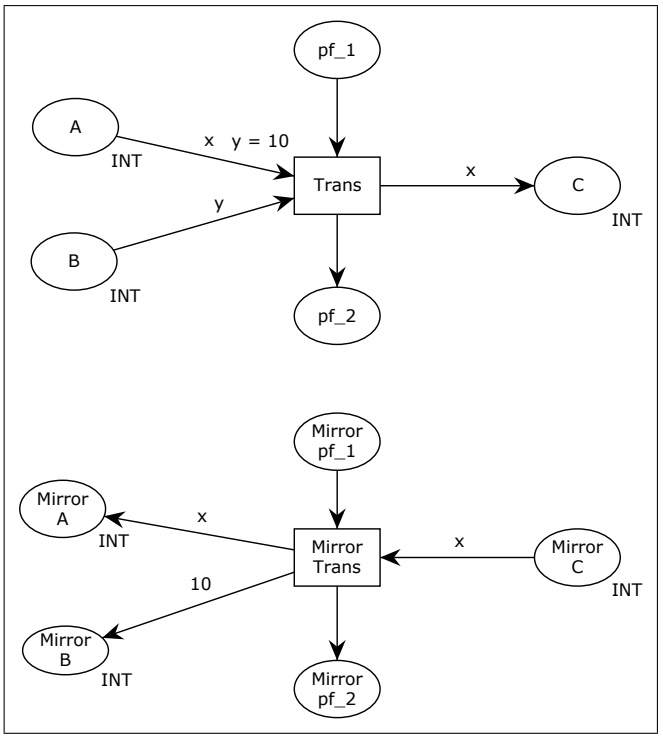

Figure 3.10: Transition with a Constant as Guard and Mirror

flow

5. A_1 are arcs which calculate a result before sending it to the Mirrored Client. (Item 5 in the introduction of this chapter)
6. We define A as: $A = A_p \cup A_d \cup A_1$ (This is the same A as in the tuple defined in 2.4, see above)
7. Σ is a finite set of non-empty colour sets.
8. V is a finite set of typed variables such that Type $[v] \in \Sigma$ for all variables $v \in V$.
9. $C : P \to \Sigma$ is a colour set function that assigns a colour set to each place.
10. $G : T \to EXPR_V$ is a guard function that assigns a guard to each transition $t \in T$ such that Type $[G(t)] =$ Bool
11. $E : A \to EXPR_V$ is an arc expression function that assigns an arc expression to each arc $a \in A$ such that Type $[E(a)] = C(p)$, where $p \in P$ is the place connected to the arc a
12. $I : I \in P$ is the set of Input Places.
13. $O : O \in P$ is the set of Output Places.
14. M: Is the mirroring rule per Arc. We define the mirroring rule M as a function that assigns a mirror rule to each arc.

Such that:

15. $P \cap T = \emptyset$.
16. $I \cap O = \emptyset$.
17. $(A_P \cap A_d) \cup (A_P \cap A_1) \cup (A_1 \cap A_d) = \emptyset$
18. The mirror rule M is an integer with the following meaning:

    -1 The arc is inverted during mirroring. $M(a) = -1$ when $a \in A_d$

    0 The arc is removed during mirroring. $M(a) = 0$ when $a \in A_1$

    1 The arc is not changed during mirroring. $M(a) = 1$ when $a \in A_p$

Note. Inverting an arc does not only change the direction of the arc, it also changes the arc expression function E and the guard function G. This definition has a minor redundancy by defining both M and the 4 types of A, this is to assist in the explanation.

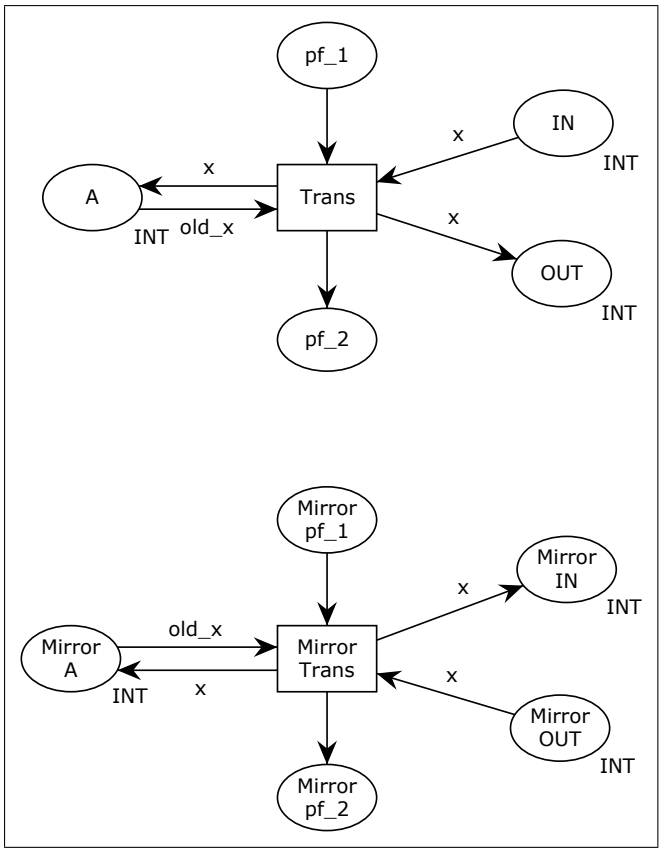

Figure 3.11: Transition with in and out var and Mirror

3.5 Logical groups of Places, Arcs and Transitions

In the mirroring process, we separate 2 types of (token) flow through the CPN. Arcs are always part of -Client Server Data flow- or -Process Flow, but this restriction does not apply to Transitions and Places.

The next examples are referencing figure 3.12

- Process Flow. The Places, Arcs and Transitions that control the order the Petri Net is executed and the order in which communication between client and server occurs. The Process Flow contains 1 token, and it needs to be a closed loop. Example: ”flow 1”, ”receive command”, ”flow 2”, ”send confirmation”, ”flow 3”, ”send result” and the connecting arcs. It usually only uses a black token but may use a coloured token if the token contains data which is relevant for the synchronisation of the client and the server (the ”id” tokens).
- Client Server data flow. The Places, Arcs and Transitions that transport data from one place

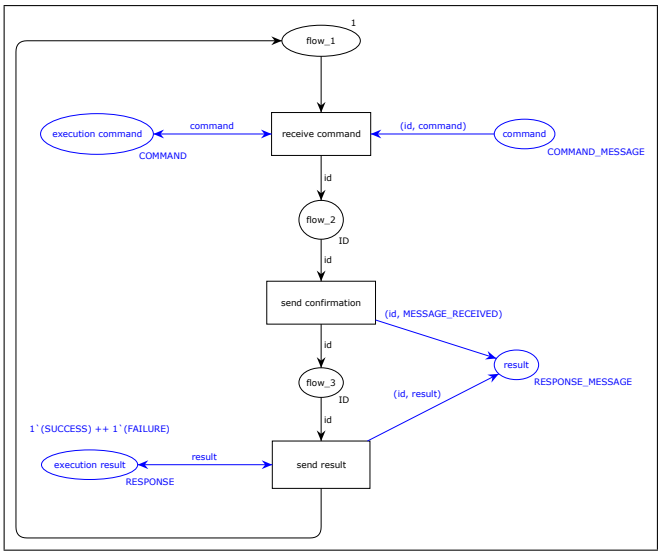

Figure 3.12: Command Server

to the Client Server Input/Output Place and vice versa. Example: Command”, ”receive command”, ”Execution Command” and the connecting arcs.

Chapter 4

Methodology and Tools

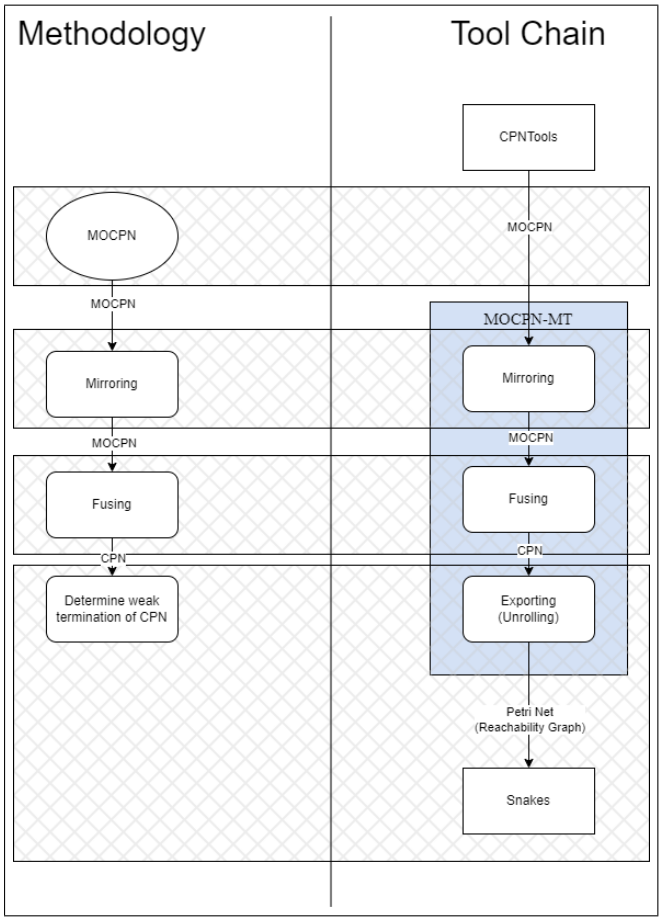

Figure 4.1: Methodology and Tool chain

The objective of this thesis is to identify a way to verify if an open-coloured petri net is weak terminating by its design. We do this by Mirroring, fusing the OCPN with its Mirror and then checking if the resulting net is weakly terminating.

4.1 Methodology

The methodology starts with a MOCPN, which we defined in the previous chapter.

Figure 4.1 on the left side:

1. We model the component or interface as a Mirrable Open Coloured Petri Net,
2. Mirror the MOCPN
3. Fuse MOCPN with its mirror by merging the connecting places this creates a fused CPN
4. Calculate the weak termination property

4.2 Alternative

In the work of Bera [11] and Hildebrand[2], this process has been investigated for Open Petri Nets. Since a CPN can be turned into a Petri Net, this gives the option to use their methods to do this analysis. When we first turn an OCPN into an OPN the rest of the process is difficult to impossible to be scrutinised visually or mathematically, while this is relatively easy when we stay with Coloured nets as long as possible. It also does not result in CPN that can be used as a guide for designing/developing clients. During unrolling, the size of the net may expand exponentially, so an OCPN with four transitions, may turn into an OPN with one million transitions depending on the number of colours in the different colorsets. Another advantage of using a MOCPN is that this makes it easier to validate the OCPN for specific use cases. In that case, the relevant tokens for the Use Case can be introduced into the merged CPN, in theory, this is still possible in the PN but requires an understanding of the unrolled PN and which states in the PN relate to the Use Case.

4.3 Tooling

To assist in the process we developed a tool that does most of the work for us. The MOCPN Mirroring Tool (MOCPN-MT) takes a MOCPN as input. We

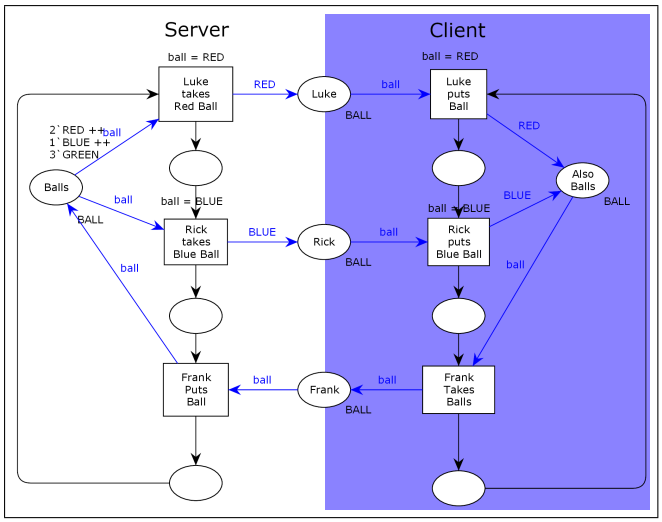

Figure 4.2: Moving Balls

create a MOCPN by using CPN tools and colouring every place/transition and arc which is part of the data flow Blue, any other colour will be regarded as part of the Process Flow or Internal data flow.

4.3.1 Tool selection

During the investigation which tools are the most useful to implement this process, we settled on CPN-Tools, Snakes and PNaT. There is a short description and links to these tools and the other ones we looked at in Appendix B
We use CPN-Tools since this is a well-known CPN editor which allows us to colour the arcs. The use of Snakes has the advantage that we can create the whole tool in Python, and because of this have full control over every part of the mirroring process. We use PNaT since this is a proven tool for calculating Weak Termination of a Petri Net.

4.3.2 Tool Chain

The full toolchain is shown in Figure 4.1 on the right side, and the corresponding part of the methodology on the left side. We use the Moving Balls from Chapter 3 for visualisation (repeated here as Figure 4.2)

CPNTools

In CPN Tools we create a MOCPN like the Server side of figure 4.2. CPN Tools does not have direct support for MOCPN, so we use blue arcs to mean $M = -1$, red arcs (not shown) to mean $M = 0$, and black lines to mean $M = 1$ (see the definition of a MOCPN, Chapter 3.4), M is defined in rule 18). We also assume that any place with only input arcs is an Output Place, and any place with only output arcs is an input place.

Process of MOCPN-MT

1. Import the Mirrorable Open Petri Net created in CPN Tools/
2. Take M from the definition of the MOCPN. (Which we defined with a color in CPN Tools)
3. We Mirror each transition in the Petri Net, by inverting the Arcs and Guard-Components for $M(a) = -1$ (including creating or removing Guards-Components), removing Arcs and Guards-Components where $M(a) = 0$ and keeping the Arcs and Guard-Components for $M(a) = 1$ and combine the Guard-Components calculated back into a Guard. (This gives us the Client side of figure 4.2)
4. We fuse the Original Net with the Mirror. (This gives the full figure 4.2)
   - We have to make sure that the Places, Transitions and Arcs have unique names, to make sure the process does not confuse the versions in the original and the mirror. This does not apply to the Input and Output Places since these are fused (I use N1 and N2 to differentiate Transitions and Places and n1 and n2 to differentiate Variables)
   - The Input and Output Places of the Original are fused with their Mirror. After merging an Input Place has Original Arcs going out and Mirror Arcs coming in, and an Output Place has Original Arcs coming in and Mirror Arcs going out.
5. Calculate State Space using snakes and then Export the State Space to PNaT format

PNaT

1. Import the created file from MOCPN-MT
2. Calculate the properties (PNaT)

3. Check the weak termination property of the fused CPN

Note. In MOCPN-MT we immediately create the state space of the CPN using Snakes instead of unrolling, since this simplifies the PnAT analysis. Unrolling means that all possible values of the different colour sets in the CPN need to be mapped to their target PN black tokens, which can easily cause a state space explosion. By using the state space we only take the coloured tokens we assign for verification in consideration to calculate the Petri Net which serves as input to PNaT.

Chapter 5

Tool Verification

To verify the functionality of the mirroring process, we use a couple of reasonably simple nets, for which we can still draw the diagrams. We show that the mirror behaves as an appropriate client. We start with a version that does not contain a process flow and show why including a process flow is important but not always required for the functionality of the net. After this, we look at what happens when we break the MOCPN to show that the methodology identifies when a net is not weakly terminating or has an unbounded state space. We are following the steps of figure 4.1 in the previous chapter.

5.1 Introduction

We have a component in a component-based system that can be turned on and a balance can be assigned to it from another component. In this case, we are only modelling an interface, we do not further specify what happens with the balance in the component. We also define a rule, that a balance can only be set if the component is turned on.

We are going to model this in 4 ways, to highlight the different aspects of the behaviour of the process, to clarify the reason for the process flow, and to show what happens when it is applied to a component which does not behave as intended.

5.2 Verification Experiment 1

5.2.1 Objective

Demonstrate what happens when we use a correctly defined OCPN, without using a process flow to keep original and mirror synchronised.

5.2.2 Experiment

We first use CPN Tools to model a MOCPN, which is shown in figure 5.1. We import the MOCPN into the mirroring tool (MOCPN-MT). To show that the import is correct, we output the OCPN from MOCPN-MT using snakes. This is shown in figure 5.2 and when compared with the original MOCPN we see that they show the same Petri net. We let MOCPN-MT do the mirroring, fusing, and calculating the state space. The fused net is shown in figure 5.3. Snakes hangs when trying to create a diagram for the state space, but we can export it to PNaT. PNaT can also not generate a diagram for the state space, but by using the PNaT version which excludes diagram creation, it does identify that this net is weakly terminating, as expected.

5.2.3 State space explosion

Since we can not visually inspect the state space, we analyse the state space by looking at the file generated for PNaT. The data in this file shows that the state space has 3242 states and 16215 transitions between these states. This explains why neither Snakes or PNaT can visualize it.

5.2.4 Findings

The methodology can handle a correctly defined OCPN which does not contain a process flow, but this can result in state space explosion because tokens can move unrestricted. This confirms that the methodology can handle Petri Nets which has multiple tokens going through it.

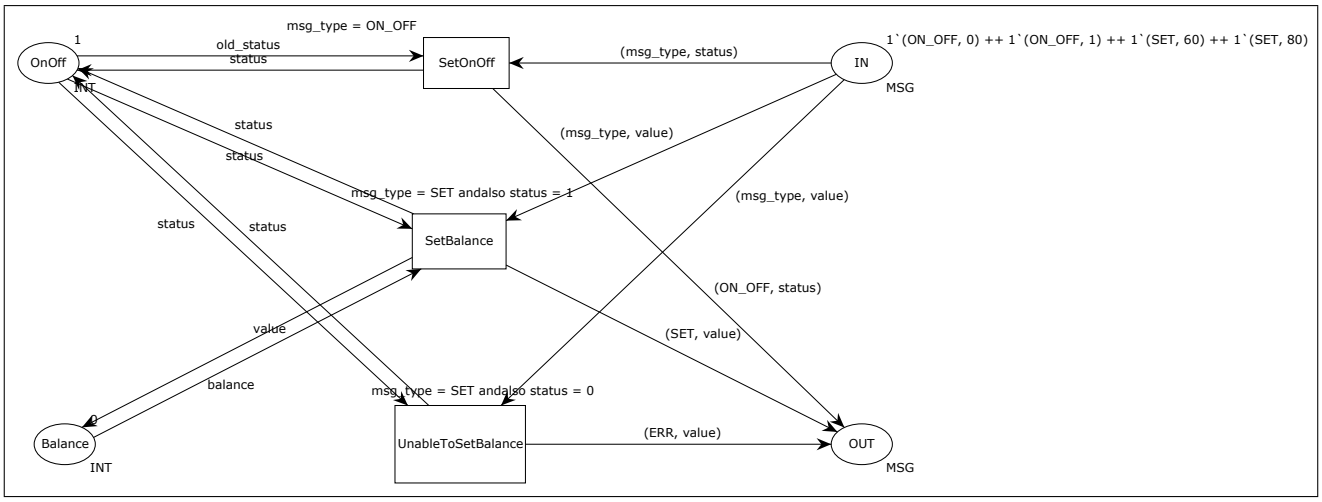

Figure 5.1: Verification 1: Model

5.2.5 Resolving the state space explosion

This example creates a relatively large state space. If we would scale up the Petri Net, it could result in a state space explosion. The reason for this is, that we don't include a process flow. So let's add this. This is shown in Figure 5.4. We added a transition to the net "Ready" and included this in a process flow. This makes sure both nets are synchronised, since if there is only one transition between IN and OUT, putting the transitions in a process flow would not have any effect when the verification tokens are directly available to the transition. The original net waits for the mirror to be ready before reading the next token. In the next example, we will explore a variant of this net, which has a smaller state space, but similar behaviour. There are alternative ways to prevent verification tokens from being available to the server before the client is ready, but these would require the client to have additional logic to do this, and the client would not be created with only the mirroring rules.

5.3 Verification Experiment 2

5.3.1 Objective

We verify an example OCPN, with a process flow to synchronising original and mirror. We expect to see weak termination and not an unexpected state space size.

5.3.2 Experiment

We use the net as described in "Resolving the state space explosion" of Verification Experiment 1 5.4. In this case, we want the client to update its Balance and OnOff places to the same values as the server. The fused net with process flow is shown in figure 5.5. Because we want the client to get the same values as the server, only the arcs connecting to the IN and OUT places are mirrored. The client first sends a READY msg to the server, after this, a token of IN is processed by the transitions of the server, and this token is sent back to out. If it is an SET msg, then the client Balance is updated, if it is an ERR msg, the client Balance is not updated, if it is an ON_OFF msg, OnOff is updated. The token is put back in IN for further verification, note that the ERR token is turned back into a SET token. After both Server and Client have finished, the Clients Ready transition is enabled, and the Servers Ready transition is waiting for a READY msg.

Generating the Reachability Graph creates the figure 5.6. Analyzing this with PNaT shows that it is weakly terminating, as expected.

5.3.3 Findings

Using a process flow to synchronise the original net and its mirror will result in a smaller state space, and assures that tokens do not move through the net unrestricted.

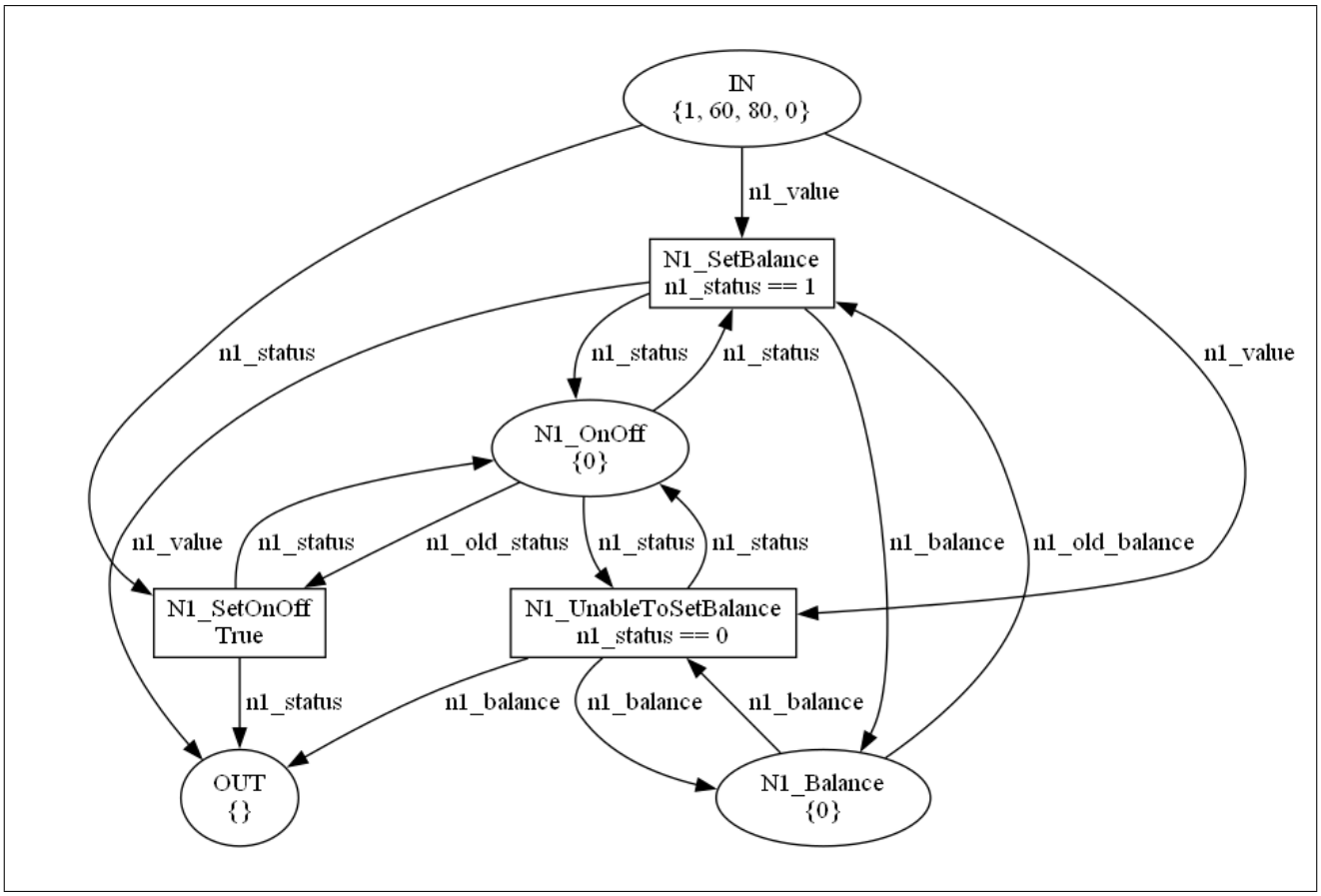

Figure 5.2: Verification 1: Example Snakes render of Model

5.4 Verification Experiment 3

5.4.1 Objective

In this experiment, we want to see if a deadlock can be identified, by creating a MOCPN which we expect to not be weakly terminating once fused with its mirror.

5.4.2 Experiment

If we decide that there is no need to inform the client of the fact that the balance can not be set, by not including an Arc connecting UnableToSetBalance with OUT, we get figure 5.7. We still use the Process Flow connecting the Model with its Mirror. This results in the Reachability Graph shown in Figure 5.8. This version is not weakly terminating, since it consumes tokens when UnableToSetBalance is triggered.

5.4.3 Findings

As we can see, this net is not weakly terminating, it has multiple end states in the Reachability Graph (the red states). These end states are deadlocks, and the Petri Net can not return to its original state.

5.5 Verification Experiment 4

5.5.1 Objective

Identify what happens when a transition is completely free to trigger, without a process flow to make sure that the original and mirror stay synchronised.

5.5.2 Experiment

The experiment states with figure 5.9 for the fused net. This experiment does not include a Process Flow, and it also does not include an Arc connecting UnabletToSetBalance to out. When calculating the state space for this, it is shown to be unbounded (Calculating the state space takes an unreasonable amount of time). This happens because with the removal of the Arc, the Transition UnableToSetBalance in the Mirror can now always trigger.

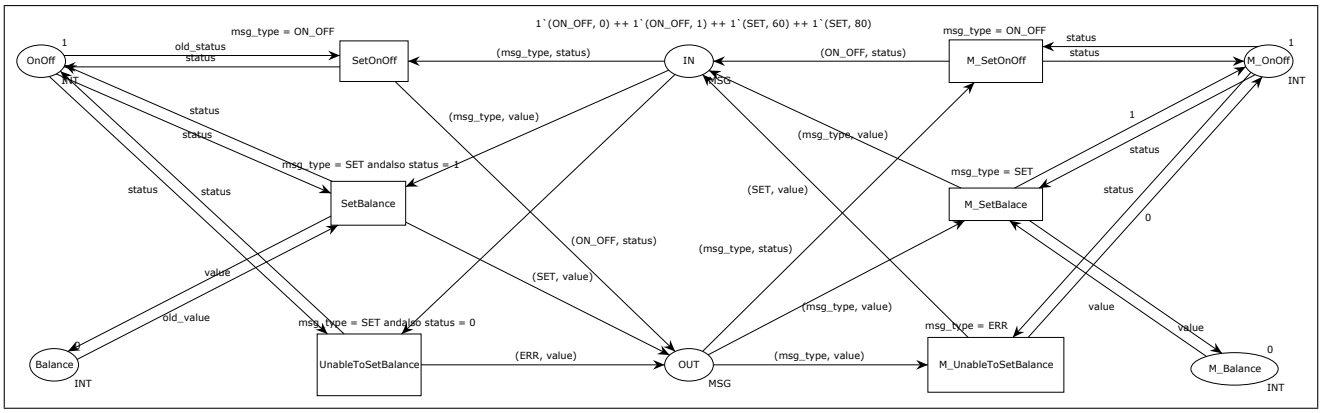

Figure 5.3: Verification 1: Fused Net

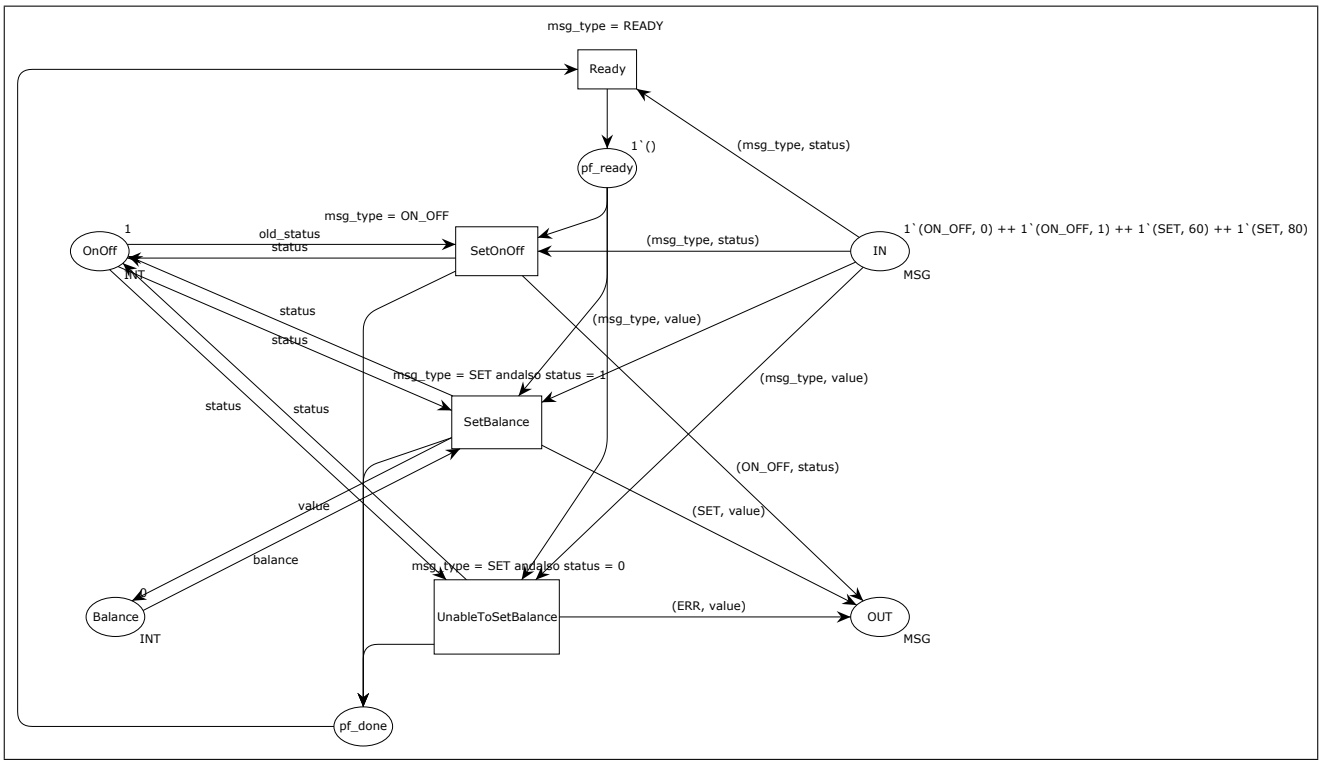

Figure 5.4: Verification 1: Resolving state space explosion

5.5.3 Findings

If a transition can always be triggered and is not restricted by the process flow, a state space explosion can occur. In this case, this results in an unbound state space, because not all tokens created are eventually consumed.

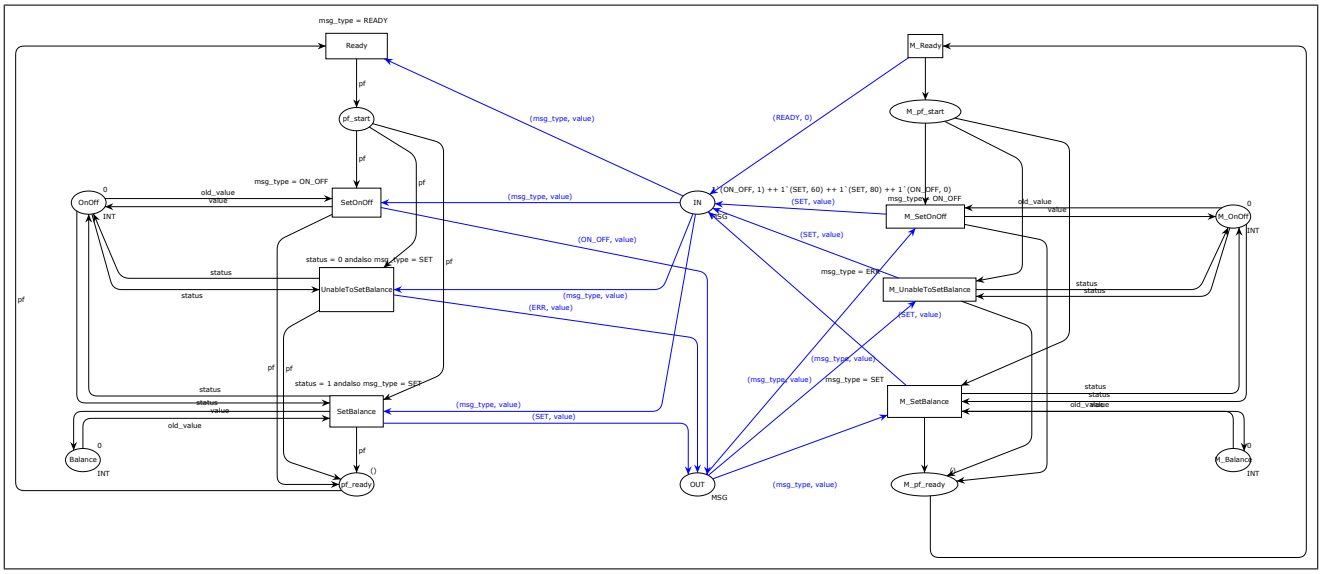

Figure 5.5: Verification 2: Fused Net

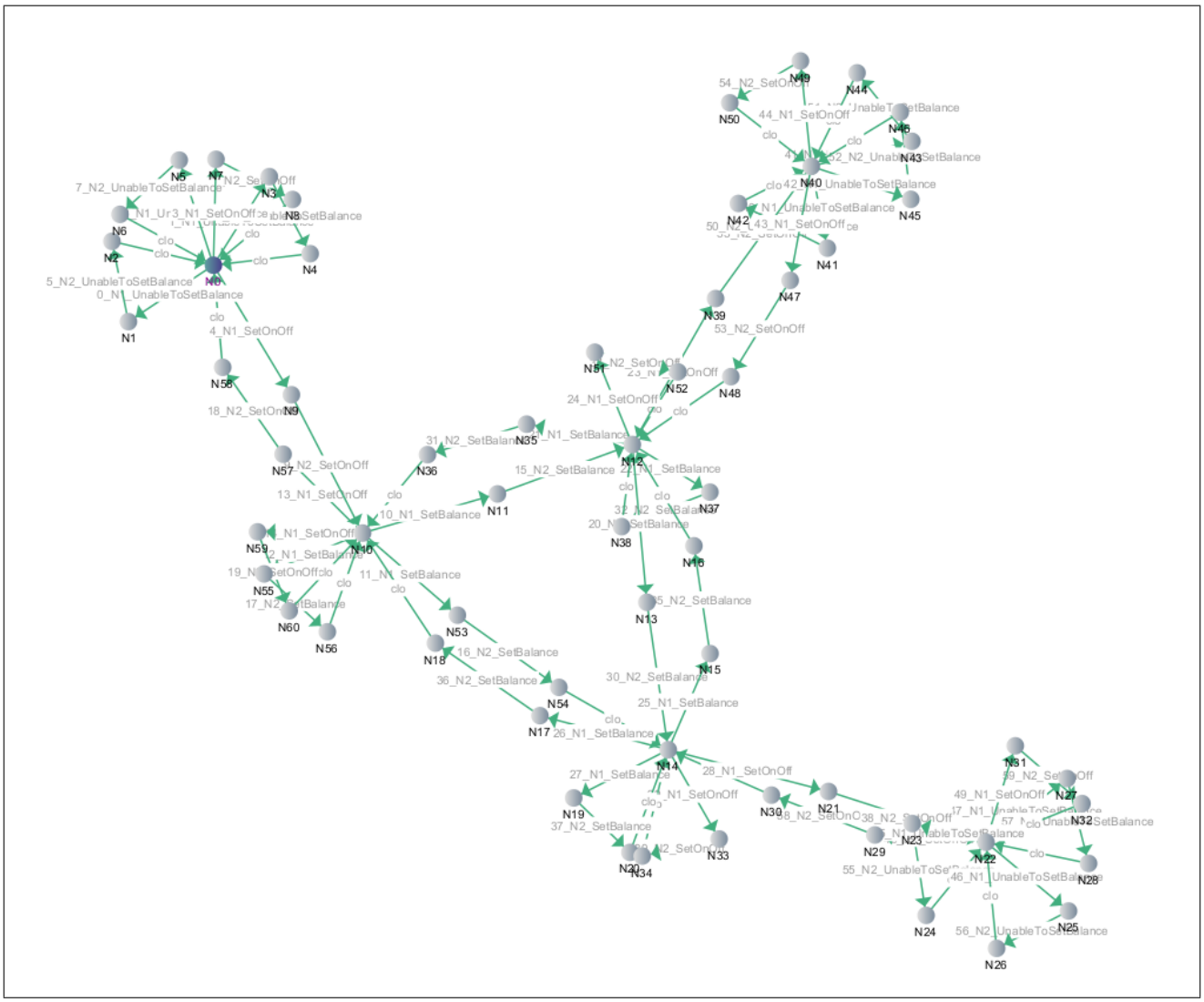

Figure 5.6: Verification 1: PNaT Reachability Graph

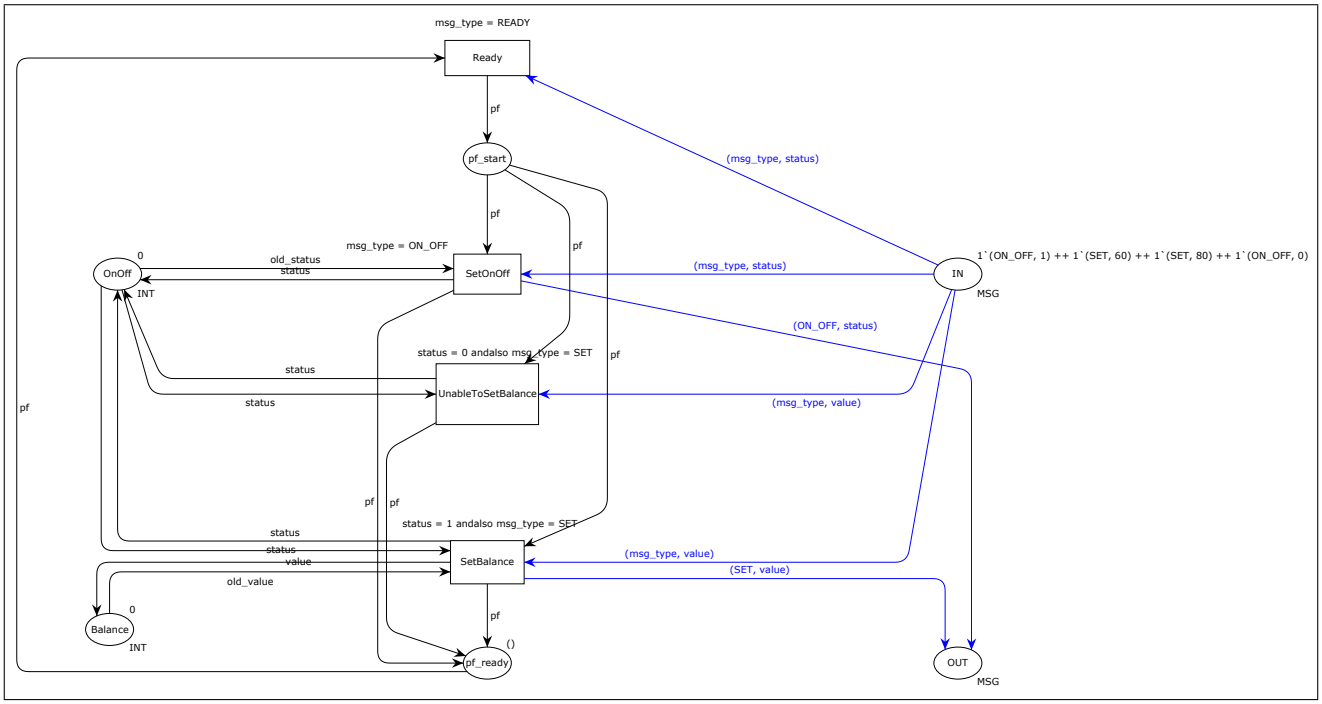

Figure 5.7: Verification 3: Model

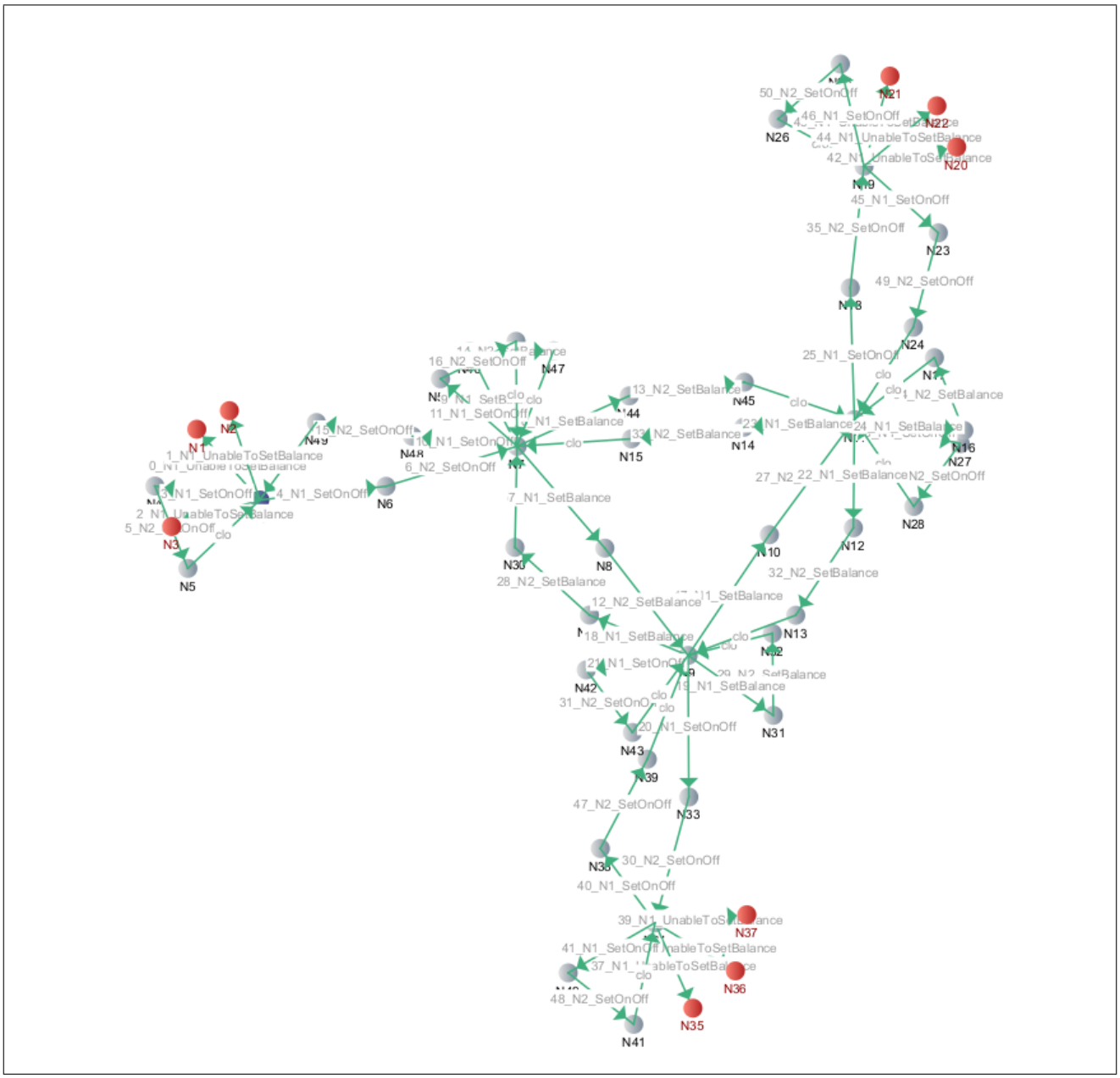

Figure 5.8: Verification 3: PNaT Reachability Graph

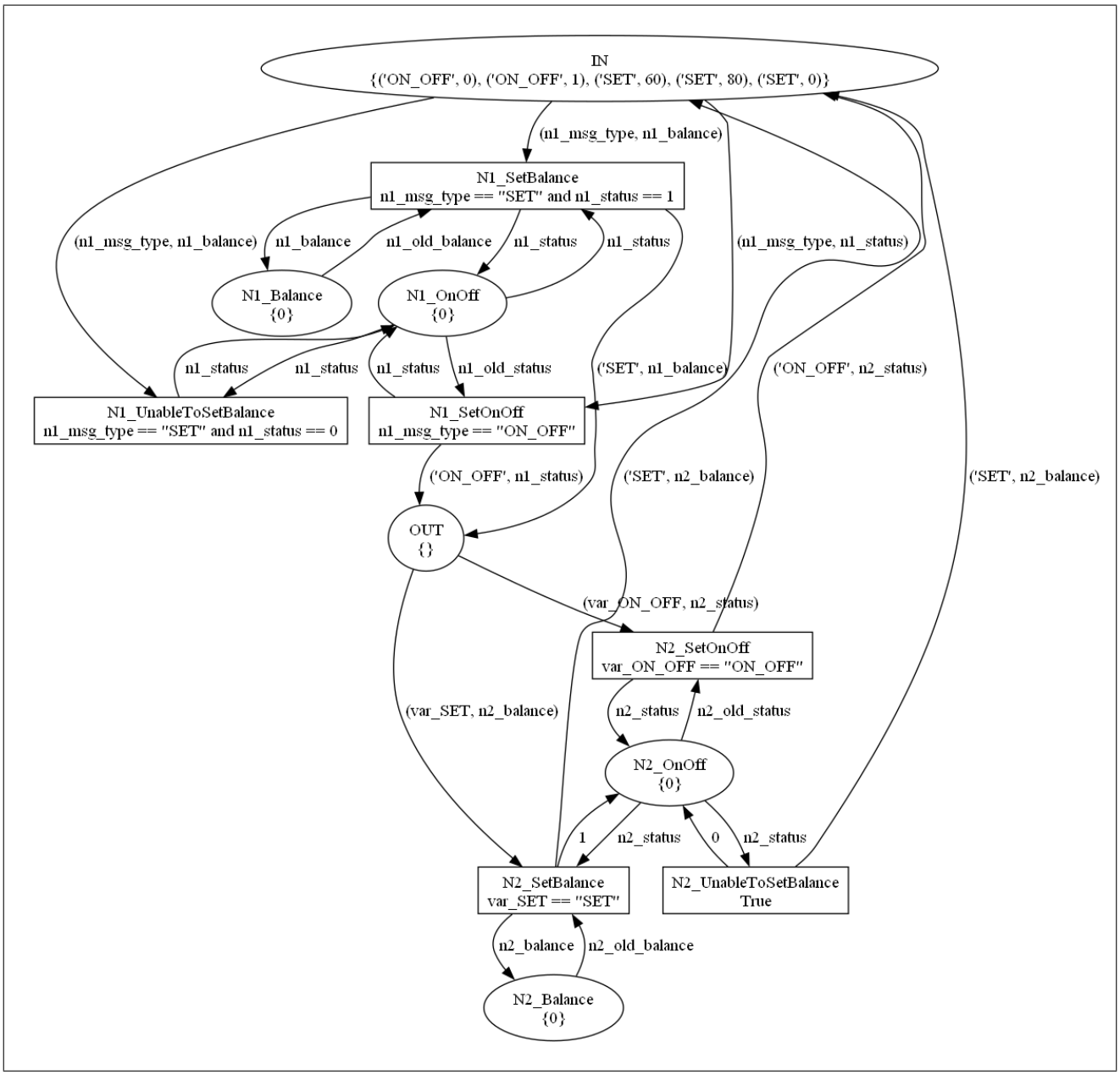

Figure 5.9: Verification 4: Fused Net by Snakes

Chapter 6

Related Work

The process of validating Components in a Component-based system has been the subject of multiple studies, like: "Compatibility of Software Components - Modeling and Verification" [18], "SaveCCM: An Analysable Component Model for Real-Time Systems" [4], and "Interface Theories for Component-Based Design"[19]. These focus on strict frameworks, strict component modelling languages with clear semantics and the relations between CBS and Interfaces between components. Some of these assume that the Interface of the components are already defined, and can be compared between components. We assume that the Interfaces of other components are not available and analyse the model earlier in the process to check if the Interface will work. Some ignore the relevance of Data as part of the execution of a Component, whereas we focus on Data guiding part of the execution.

There are also papers which use the model of a Component as a way to generate runtime validation by creating a monitoring tool [20]. But this would not be right for critical systems, getting an error message after something is wrong may be too late. Even when the issue is found during integration of the components, correcting this may be very expensive, and may require a redesign of the component. The approach we take in this thesis can be applied much earlier in the lifecycle of a component, preferably already during the investigation and design phases.

The papers of Bera [11] and Hilbrands [2] discuss similar techniques as we use in this Thesis to identify weak termination using Petri Nets (as opposed to Coloured Petri Nets). And also defines a specific type of OPN called Port Net, which enforces Weak Termination by design. But Petri Nets do not deal with data as part of the Process Flow (the order in which Transitions are executed). The addition of Data (Colours) to the net, makes that the techniques discussed here can not be applied as is. An OCPN can be unrolled into an OPN, but we need the Mirroring Rule (M) included in the MOCPN to make sure that the client works as intended. The MOCPN definition does not have an equivalent OPN at this time. It is also questionable if this would be a useful extension to an OPN. Unless an OPN is unrolled from an OCPN, there is no need for this, and if we already have a MOCPN, then the methodology discussed in this thesis would be more practical.

During the preliminaries, we discuss Inversion 2.4.2. [14], [15], [16] and [17] look at inversion of a CPN. Mirroring and Inversion use the same techniques to calculate Guards based on Arc Inscriptions and Arc Inscriptions based on Guards. We identified that the arcs which are part of the Process Flow in the Mirror needs to be pointing in the same direction as in the original. The arcs which are part of data movement get inverted in the mirror. So a Mirror is not the same as an Inversion, but they are similar. The function of an Inverse is completely different from the function of a Mirror. An Inverse allows a CPN to be executed backwards. Inversion of an OCPN and then merging the two nets is not very useful, since the Inverse does not function as a verification client, except in some edge cases. At the same time, Mirroring a CPN (as opposed to an OCPN) is also not very useful, since Mirroring specifically creates a client which can exercise the original.

Chapter 7

Conclusion

To summarise the findings of the thesis:

- It became evident that the subject of verification client generation by using Mirroring of an OCPN is much larger than anticipated.
- An Open Coloured Petri Net can be mirrored to generate a client to exercise the Petri Net, to calculate a Reachability Graph and validate for Weak Termination.
- There are restrictions and rules for generating the (Mirrored) Verification Client, but we did not identify all of them.
- Mirroring is a partial Inversion of a net, and it has overlapping, but also distinct rules from Inversion.
- The use of a MOCPN allows it to be automated fully.

We will now answer our original research questions, the limitations of this methodology as applied in this Thesis, and future work to expand on the subject and possibly remove some of the restrictions we applied in this thesis.

7.1 Answers to Research Questions

How can a Component or the Interface of a Component, which exchanges data with other components in a Component-based system, and this data changes the behaviour of this component, be verified for weak termination?

The problem of verification of a component or interface is being investigated from several different angles by researchers, but there is no perfect solution. The most common problem is state space explosion during analysis. By using a Coloured Petri Net, and restricting the verification to verification Tokens, we can verify weak termination, with only moderate or no state space explosion (depending on the Net).

Since a component or interface is not a complete Coloured Petri Net, but one which has a boundary where tokens enter or exit the Petri Net, it means that the weak termination can not be fully established. Only looking at the skeleton of the Petri Net, does not capture the interaction with other components. Identifying which transition to trigger is not affected by the Data (Tokens) coming from other components. Not knowing exactly which transition to trigger does not give a complete picture to establish the weakly terminating property. Therefore we needed a new class of CPN, which we define as an OCPN (3.1).

After this, we needed a way to create a closed CPN from this OCPN. We found that using Inversion (2.4.2), we could create a counterpart which can be used with the OCPN. However, a fully inversed OCPN does not always behave correctly to be used to complete the CPN. The OCPN and its mirror do not synchronise their actions, and this is important. This identified another issue, how do we know exactly in which order the Transitions needed to be triggered? There was not enough information in the OCPN to reliably detect in which order the Transitions need to be triggered (the Process Flow). We introduced the Mirroring Rule (M), to keep the original and calculated version synchronised, where $M = 1$ identifies the Process flow. Therefore this information needed to be added to the original OCPN, and the MOCPN was defined to address this (3.4).

We also restricted ourselves to mirroring transitions, not allowing calculations on arcs or comparators (except for equals) on the guards.

When we have the MOCPN of a component or its interface, we can generate a mirrored version of this MOCPN. Fusing these two MOCPN creates a CPN which can be verified for weak termination.

Can we identify rules to generate a component model from the original component which, when fused with this component model can verify for weak termination?
The need to manually mark the arcs with the Mirroring Rule (M) makes the automatic application to any Open Coloured Petri Net difficult. However once a MOCPN is made it can be mirrored and there are specific rules and constraints for both the MOCPN and the mirroring methodology (4.1).

We identified the base rules needed for this to function. The MOCPN can only use a subset of the guards and arc inscriptions compared to a CPN. In this Thesis, we restrict this severely. Tuture work on this topic may show that guards and Arc Inscriptions can be less restricted, or another type of OCPN could be created which gives more freedom. When we restrict ourselves to this subset, we can calculate the guard of a Mirrored Transition based on the arc inscriptions going to this Transition. We can also calculate the Mirrored arc inscription(s) going from this Mirrored Transition based on the Guard of the Transition.

We identified that the arcs which are part of the Process Flow in the Mirror needs to be pointing in the same direction as in the original. The arcs which are part of data movement get inverted in the mirror. There may be a need for some arcs to be removed in the Mirror (to assure certain tokens are only generated in the original and then send to the mirror).

Are these rules strict enough to be implemented as an algorithm? How would a program applying this algorithm be defined?
After identifying the rules needed, we created the MOCPN-MT program, which applied the mirroring rules, and then fusing the two MOCPN to create on CPN for verification. With the addition of the Mirroring Rule (M), the rules are strict enough to automate the methodology.

Combined with the full tool chain (4.3.2) it can check for weak termination based on the MOCPN of the original, and can generate the needed Mirror based on strict rules. We however did not identify strict rules for the tokens needed for verification.

7.2 Future Work

The subjects listed in the limitations are interesting for further research.

7.2.1 Expansion of Guards and Arc Inscriptions

Mirroring arc calculations is problematic. These calculations have a clear inverse, but applying this inverse to a mirror affects the functions of the mirror. We have an arc that increments a value to be used as an identifier f(x) y = x + 1 then the inverse is f(y) x = y - 1, which would set the identifier in the mirror to x and not to y. From this questions arise: Can a calculation be mirrored? The same can be said for mirroring guards with comparators. Can we Mirror Guards which contain "or", "less than", "greater than" and "not equals", how can we identify the outgoing values of the Transition?

7.2.2 Full Petri Net Mirroring

In this study, we Mirror every Transition separatly from every other Transition. But the paper [16] which looks at Coloured Petri Net Inversion does not limit itself to one transition for inversion, but combines information of the whole net. Can a similar approach can be used for mirroring? Using a similar approach may be able to create Mirrors which do not have all the restrictions (On the Guards and Arc Inscriptions).

7.2.3 Mirrorable by Design

In Black Token Petri Nets we have the variant of a Port Net, which has specific rules which, when fused with its mirror, are always Weakly Terminating [11]. Can similar rules be found for a Coloured Port Net?

7.2.4 MOCPN-MT

Although the MOCPN-MT program allowed us to validate this Thesis, it is not a mature tool. The parser for the Arcs and Guards is temperamental, and Snakes does not handle integers and black tokens properly. Snakes also does not use ColourSets.

This creates unexpected and difficult-to-identify exceptions in the tool.

It would be best to write a new version of MOCPN-MT which addresses these issues and does not rely on Snakes. It would also be nice if the tool could check for the Weakly Terminating property in stead of exporting to PNaT.

7.2.5 Complimenting Methodologies

In Related work 6 we mention [20]. It may be possible that Mirroring can be used to generate a client. The server can then be run with this client and the Monitoring technique described in the paper can monitor the system while it is being exercised by the client. Allowing a monitoring check to be done earlier in the development process. There is however the problem that we define our server as a MOCPN while [20] uses the BIP (Behavior, Interaction, and Priority) framework [21] to define the server.

Appendix A

Identifying M

M defines the function of the Arc, this may not always be derivable from an OCPN definition. The following steps can be used to identify M but keep in mind that the function of an Arc is not always clear, in that case, M should be identified by analysing the function of the Arc.

A.1 Arcs from or to an interface place

Interface places are the places where the Client and Server connect: $o \in O$
All arcs connecting to Interface places are Client Server Data Flow Arcs and have $M(arc) = -1$
$M(arc) = -1$ when $arc = ({}^\bullet o, o)$ or $(o, o^\bullet)$

A.2 Process Flow Arcs

When executing the Petri Net and identifying the firing sequences $T_1 \rightarrow T_2 \rightarrow T_3$, the Process Flow Arcs are arcs which connect T_n to T_{n+1} with only one Place between them.
If the order of execution is not known (because a client is needed), the OCPN can be transformed as follows to find it:

- Remove all interface places $o \in O$
- Replace all tokens with Black tokens and all Coloursets with the Black Token Colourset
- Remove all guards
- Execute this Petri Net. The Process Flow Arcs are arcs which connect T_n to T_{n+1} with only one Place between them.

A.3 Other Client Server Data Flow Arcs

For every Transition t connecting to an Interface place o, there is now an M for most of the connecting arcs. The other arcs are likely Client Server Data Flow Arcs when:

- The data of the arc, is coming from or going to the arc connecting to o
- The data of the arc, is defining the data, or is defined by the data of the arc connecting to o
- The arc is incoming or bidirectional, and the arc connecting t to o is outgoing.
- The arc is outgoing or bidirectional, and the arc connecting t to o is incoming.

These arcs have $M(arc) = -1$
If the data of the arc is only used by the guard function of t, this arc is most likely not a Data Flow Arc $M(arc) = 1$.

A.4 Other Arcs

The other arcs have to be checked for their function and how they are expected to behave in the Mirror. The Process Flow and the Client Server Data flow may not have any other joint transitions as the ones above.
$M(arc) = 0$ is used for arcs with tokens calculated by the Server, which the client is not allowed to calculate itself. For instance, a unique identifier for the communication.

Appendix B

Tools Overview

The following tools have been looked at as part of the selection process. Quite a lot were not suitable for the purpose we needed, but they are listed here for completeness.

- Snakes

  A net algebra kit for editors and simulators

  https://snakes.ibisc.univ-evry.fr

- MCC

  A Tool for Unfolding Colored Petri Nets

  https://github.com/dalzilio/mcc

- CPN Tools

  A tool for editing, simulating, and analyzing Colored Petri nets

  https://cpntools.org

- Neco

  Compiles a Petri Net into efficient C code which is then compiled into a native library that can be loaded as a Python module.

  https://github.com/Lvyn/neco-net-compiler

  https://code.google.com/archive/p/neco-net-compiler/wikis

- ABCD

  A language for formal modelling and analysis

  https://hal.archives-ouvertes.fr/hal-01352028

- ePNK

  The ePNK is a platform for developing Petri net tools based on the PNML transfer format.

  http://www.imm.dtu.dk/~ekki/projects/ePNK/index.shtml

- MISTA

  A Tool for Automated Test Code Generation from High-Level Petri Nets

  Unable to locate online

- Petri Net Kernel

  An infrastructure for building Petri net tools.

  https://sourceforge.net/projects/pnk/

- GreatSPN

  a software package for the modelling, validation, and performance evaluation of distributed systems using Generalized Stochastic Petri Nets and their coloured extension, Stochastic Well-formed Nets.

  https://github.com/greatspn/SOURCES

- MARIA

  a reachability analyzer for concurrent systems that use Algebraic System Nets (a high-level variant of Petri nets) as its modelling formalism.

  http://www.tcs.hut.fi/Software/maria/index.en.html

- Tapaal

  A tool for Verification of Timed-Arc Petri Nets

  Website: https://www.tapaal.net/

  Documentation: https://leanpub.com/tapaalusermanual/read

- Snoopy

  A software tool to design and animate hierarchical graphs, including Petri nets.

  https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Snoopy

  - Marcie

    a tool for qualitative and quantitative analysis of Generalized Stochastic Petri nets with extended arcs.

    https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Marcie

  - Charlie

    an extensible software tool to analyse (extended) place/transition nets

    https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie

  Note: The Snoopy software does not contain a license, which may be problematic for commercial use. Marcie has the statement that it is "free for non-commercial use only" on the website without further details.

Appendix C

Tool Details

C.1 Import CPN from CPN-Tools

For this thesis, we use CPN-Tools to create the OCPN which we supply to MOCPN-MT. In CPN-Tools we use different colours of the arcs to define whether an arc is a Process Flow Arc (black) or a Data Flow Arc (blue).
CPN-Tools save a Petri net in XML, we can transform this into Python code by using an XML transformation (xlst). The code to apply the xlst to the CPN-Tools save is straightforward by using the lxml library.
After the transformation, we execute this Python code to create the NetDefinition.

C.2 Calculating the Mirror

At this point, we have the NetDefinition of the OCPN, and we need to calculate the Mirrored NetDefinition.

```
Create an empty NetDefinition which we will call MirrorDef.
Copy all NetDefinition.Places to MirrorDef.Places without the Arcs.
For each Transition in NetDefinition.Transitions
    Create an empty MirrorTransition with the same id as the original Transition
    For each Arc in Transition.Arcs
        If the Arc is a Data Process Arc
            Inverse the Arc, which may include changing the Arc Inscription
            Calculate the inverse Guard function related to this Arc
            Add the MirrorArc to: MirrorDef.Arcs, MirrorTransition, the Place it connects.
        Else
            Copy the Arc
            Assign the original Guard function to the mirrored Guard function.
            Add the Arc to: MirrorDef.Arcs, MirrorTransition, the Place it connects.
    Combine all the mirrored guard functions into one, and update the Transition.
    Add the Transition to the MirrorDef.Transitions
```

C.3 Merge the Nets

At this point, we have the NetDefinition of the OCPN, and the Mirrored NetDefinition.

```
Create an empty NetDefinition which we will call MergedDef.
```

```
Copy the Interface Places to MergedDef once (From either net, but not from both)
Except for Interface Places:
    Copy all Transitions, Places and Arcs from both nets to MergedDef
        rename all places based on their origin (We prefix them with N1_ and N2_)
        rename all variables based on their origin (we prefix them with n1_ and n2_)
```

We now have a merged net where the Interface places still have the same name as in the original OCPN, and all other elements of the net have a new name identifying if they are part of the original net or the mirror net.

C.3.1 Create Snakes Net

During the creation of the Snakes Net, we do some transformations on the Guard and Annotation objects to align them with the expected input of Snakes, but aside from this, it is as simple as adding all the NetDefinitions.Transitions, NetDefinitions.Places and NetDefinitions.Arcs to the SnakesNet.

C.3.2 Create State Chart

This is a function of Snakes itself, which takes the SnakesNet as input and creates the StateChart.

C.3.3 Export for PNaT

Navigate through the StateChart, every state becomes a PNaT Place and every transition between states becomes a Transition with an incoming and outgoing arc connecting the places which correspond with the States it connects

Appendix D

Class Diagram of MOCPN-MT

We need to transform the Petri Net before having it processed by Snakes, so we need an internal representation of the Petri Net which we can manipulate. The main class in this system is the NetDefinition which holds our definition of the Petri Net as well as the generated SnakesNet and StateCharts. The NetDefinition is used for the original OCPN, the Mirrored OCPN and the Merged CPN, so during execution we will have 3 instances of NetDefinition related to the different steps in the process. We excluded the Import class from the class diagram since this is specific to MOCPN-MT which creates the OCPN. The Import class is expected to create all the objects in the diagram based on the OCPN with the exception of the StateChart and SnakesNet, this is the first NetDefinition. The other two NetDefinitions are calculated based on this.

We combine MOCPN: $\mathcal{N} = (P, T, A_p, A_d, A_n, A_1, \Sigma, V, C, G, E, I, O, M)$ (Chapter 3.4) with the Snakes notation, using $A = A_p \cup A_d \cup A_n \cup A_1$ and move the functions C, G, I, M into the elements they operate on.
Since we know that the Transition Guards and the Arc Inscriptions impact each other during mirroring, we also split the Guards into their components to simplify inversions and recombination.
The class diagram is shown in figure D.1

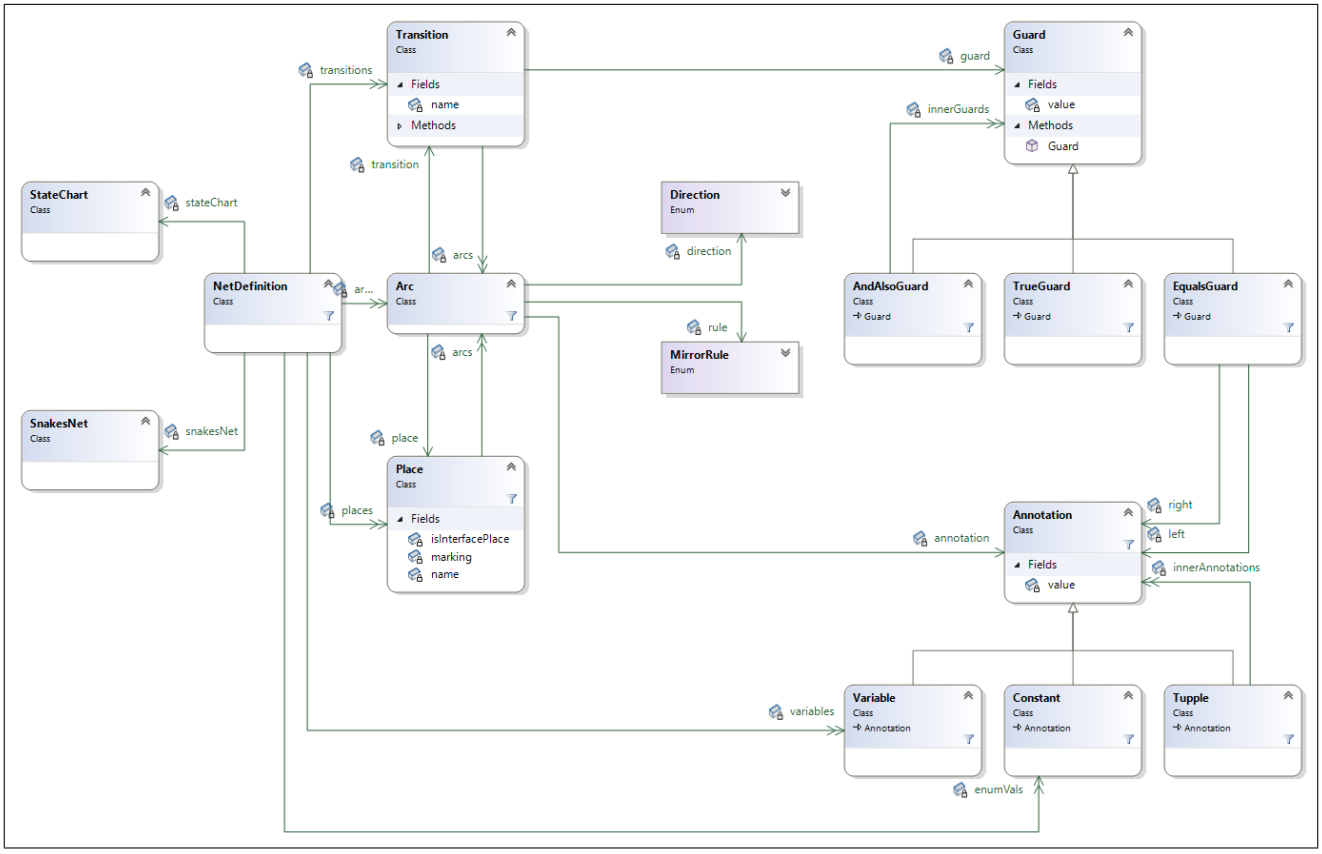

Figure D.1: Class Diagram

Bibliography

[1] Staff, "Sunk by Windows NT," *WIRED*, Jul. 1998. [Online]. Available: https://www.wired.com/1998/07/sunk-by-windows-nt.

[2] B.-J. Hilbrands, D. Bera, and B. Akesson, "Partial specifications of component-based systems using petri nets," in *CEUR Workshop Proceedings*, CEUR, 2022.

[3] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath, "Cadena: An integrated development, analysis, and verification environment for component-based systems," in *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, pp. 160–172.

[4] J. Carlson, J. Håkansson, and P. Pettersson, "SaveCCM: An Analysable Component Model for Real-Time Systems," *Electron. Notes Theor. Comput. Sci.*, vol. 160, pp. 127–140, Aug. 2006, ISSN: 1571-0661. DOI: 10.1016/j.entcs.2006.05.019.

[5] C. A. Petri, "Kommunikation mit automaten," 1962.

[6] C. A. Petri, *COMMUNICATION WITH AUTOMATA: Volume 1 Supplement 1*. 1966.

[7] C. A. Petri, "Introduction to general net theory," in *Net Theory and Applications*, W. Brauer, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 1–19, ISBN: 978-3-540-39322-1.

[8] W. Reisig, *Understanding petri nets: modeling techniques, analysis methods, case studies.* Springer, 2013.

[9] C. A. Petri and W. Reisig, "Petri net," *Scholarpedia*, vol. 3, no. 4, p. 6477, Apr. 2008, ISSN: 1941-6016. DOI: 10.4249/scholarpedia.6477.

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

[11] D. Bera, K. M. Van Hee, M. Van Osch, J. M. E. van der Werf, *et al.*, "A component framework where port compatibility implies weak termination.," in *PNSE*, 2011, pp. 152–166.

[12] W. Wu, Z. Xing, H. Yue, H. Su, and S. Pang, "Petri-net-based deadlock detection and recovery for control of interacting equipment in automated container terminals," *IET Intelligent Transport Systems*, vol. 16, no. 6, pp. 739–753, 2022.

[13] S. Dal Zilio, "MCC: A Tool for Unfolding Colored Petri Nets in PNML Format," in *Application and Theory of Petri Nets and Concurrency*, Cham, Switzerland: Springer, Jun. 2020, pp. 426–435, ISBN: 978-3-030-51831-8. DOI: 10.1007/978-3-030-51831-8_23.

[14] S. Khalfaoui, "Méthode de recherche des scénarios redoutés pour l'évaluation de la sûreté de fonctionnement des systèmes mécatroniques du monde automobile," Ph.D. dissertation, Institut National Polytechnique de Toulouse-INPT, 2003.

[15] M. Bouali, P. Barger, and W. Schon, "Backward reachability of colored petri nets for systems diagnosis," *Reliability Engineering & System Safety*, vol. 99, pp. 1–14, 2012.

[16] ——, "Colored petri net inversion for backward reachability analysis," *IFAC Proceedings Volumes*, vol. 42, no. 5, pp. 227–232, 2009.

[17] M. Bouali, J. Rocheteau, and P. Barger, "Backward reachability analysis of colored petri nets," in *The European Safety and Reliability Conference (ESREL'09)*, Taylor & Francis Group, vol. 3, 2009, pp. 1975–1981.

[18] D. Craig and W. Zuberek, "Compatibility of software components - modeling and verification," in *2006 International Conference on Dependability of Computer Systems*, 2006, pp. 11–18. DOI: 10.1109/DEPCOS-RELCOMEX.2006.13.

[19] L. de Alfaro and T. A. Henzinger, "Interface Theories for Component-Based Design," in *Embedded Software*, Berlin, Germany: Springer, Sep. 2001, pp. 148–165, ISBN: 978-3-

540-45449-6. DOI: 10.1007/3-540-45449-7_11.

[20] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem, "Runtime Verification of Component-Based Systems," in *Software Engineering and Formal Methods*, Berlin, Germany: Springer, 2011, pp. 204–220, ISBN: 978-3-642-24690-6. DOI: 10.1007/978-3-642-24690-6_15.

[21] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-time Components in BIP," in *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, IEEE, pp. 11–15, ISBN: 978-0-7695-2678. DOI: 10.1109/SEFM.2006.27.