Thesis Dissertation

SIMULATING TRUE CONCURRENT REVERSING PETRI NETS USING ASP

Loukia Christina Ioannou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

MAY 2024

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

SIMULATING TRUE CONCURRENT REVERSING PETRI NETS USING ASP

Loukia Christina Ioannou

Supervisor

Dr. Anna Philippou

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2024

Acknowledgements

I am deeply honoured to have worked with the excellent professor Anna Philippou. Her generous advice and creative thinking opened my mind to new knowledge. Throughout the year she provided me continuously with all the required information and supported me to complete my thesis. For this, I am truly grateful.

I would like to express my gratitude to my PhD. Supervisor, Eleftheria Kouppari for her valuable advice and for the willingness she showed to help me overcome challenges that arose during my research. I would also like to thank professor Yannis Dimopoulos for giving me guidance to optimize the simulator.

I also would not have undertaken this journey without my family and my dear friends. Their unconditional love gave me courage in every step that I have taken, and I thank them endlessly.

Abstract

Inspired by the complexity and elegance of Reversing Petri Nets, in our thesis we introduce the True Concurrent Reversing Petri Nets (TCRPNs). TCRPNs are a form of Petri Nets that take advantage of the benefits of reversible computation that Reversing Petri Nets provide and additionally allow the true concurrent execution of transitions. In our thesis we define TCRPNs formally and then we simulate them using the Answer Set Programming (ASP) language, Clingo. Furthermore, we introduce some optimization techniques for our Clingo code.

In addition to the simulation, we developed a tool in Pyhton3 that produces TCRPNs. This tool uses a combination of components and bridges that get attached to each other to form a big TCRPN.

Table of Contents

Table of Contents v

Table of Figures viii

Chapter 1 Introduction 1

1.1 Motivation 1

1.2 Methodology 2

1.3 Thesis Organisation 3

Chapter 2 Related Work 4

2.1 Introduction to Petri Nets 5

2.1.1 General Introduction 5

2.1.2 Formal Definition and Behaviour 5

2.1.3 Example of Application: The Candy Machine 6

2.2 Reversible Computation 8

2.3. Reversible Modelling 8

2.3.1 Forms of Reversibility 8

2.4 Reversing Petri Nets 9

2.4.1. Definition of Reversing Petri Nets 10

2.4.1. Forward Execution 13

2.4.3 Backtracking 14

2.4.4 Causal-Order Reversibility 15

2.5 Answer Set Programming 17

2.6 Clingo 17

2.6.1 Terms 18

2.6.2 Facts, Rules, and Constraints 18

Chapter 3 True Concurrent Reversing Petri Nets 19

3.1 Formal Definition 20

3.2 Forward Execution 20

3.3 Backtracking 21

3.4 Causal-Order Execution 22

Chapter 4 – TCRPNs Simulator 24

4.1 Simulating a TCRPN 24

4.2 Simulating Forward Execution 26

4.3 Simulating Reverse Execution 27

4.4 Optimizations 28

4.5 Weaknesses 28

Chapter 5– Tool for Generating TCRPNs 29

5.1 A Tool for Generating Petri Nets 29

5.2 Components 30

5.3 Bridges 31

5.4 Main TCRPN 31

5.5 Implementation Details 33

5.6 Capabilities 34

5.7 Weaknesses 34

Chapter 6 Conclusion 35

6.1 Summary 35

6.2 Challenges 36

6.3 Future Work 36

Appendix 38

GitHub Repository 38

Command Line Input and Example of Execution of the Simulator 38

Command Line Input of Tool for Generating TCRPNs 40

Bibliography 42

Table of Figures

Figure 2.1 A Petri Net modelling a candy machine. 7

Figure 2.2 The Petri Net modelling a candy machine after the fire of the leftmost transition, ‘pay 5 cents’ 7

Figure 5.1 A main TCRPN 33

Chapter 1 Introduction

1.1 Motivation 1

1.2 Methodology 2

1.3 Thesis Organisation 3

1.1 Motivation

Reversing Petri Nets (RPNs) are an extension of the traditional Petri nets with the difference that they integrate the concept of reversible computation. Recovering past states is a strong ability that RPNs have, enabling them to simulate distributed systems and correct their failures ahead of time [1].

Today, with the rapid advancement of technology, the distributed systems have undergone a perpetual evolution [2] and their size and complexity became much denser. Therefore, the need for efficient simulators that detect errors without delay increased dramatically. The question that rises is ‘how can we make a step forward and expand RPNs to achieve even further efficiency?’. We answer to this question by proposing TCRPNs. In this paper we establish an extended model of RPNs, the TCRPNs that have an additional ability that makes them good candidates for simulating systems efficiently. This advantage is that TCRPNs offer not only reversible operations, but also true concurrency, i.e. transitions are allowed to fire concurrently.

To capture the behaviour and properties of TCRPNs, we simulate them using Answer Set Programming (ASP). ASP is a programming paradigm specifically designed to solve NP-hard search problems by presenting answer sets. The simulation of a TCRPN and the exploration of its reachable states can be reduced to a search problem, making ASP an

ideal paradigm for this purpose. Thus, we use ASP to effectively simulate TCRPNs and analyse their dynamic behaviours.

Since TCRPNs are a new promising concept, it is possible to attract many researchers whose goal is to understand at a maximum level TCRPNs' benefits and capabilities, or even their disadvantages. However, a challenge that occurs when performing research, is the absence of a large number of data, in our case TCRPNs.

To strike this issue we understood that it was crucial to create a tool that produces such data. And so, we did. This tool is also a part of the thesis therefore it is explained in later chapters.

1.2 Methodology

To complete this thesis, we had to achieve several milestones.

The first goal was to be familiar with the paradigm of ASP and Clingo. ASP was a newly explored concept for us that required a lot of practice and a lot of problem solving. The language we used to write our code was Clingo. Thus, we also had to learn Clingo's elements (syntax etc.).

The second step we have taken was to collect information about Petri Nets and RPNs and understand their structure. Once we comprehended how Petri nets and RPNs are structured, it was our aim to familiarize ourselves with the concept of reversible computation and the forms of reversibility in RPNs.

When we had a complete idea about the RPNs and reversibility in general, we set a new goal. This goal was to implement the code in Clingo. That is, the code for simulating TCRPNs. Alongside development we started to form the formal definition of TCRPNs and its forms of reversibility, backtracking, and causal order execution.

The final milestone we achieved was to develop the tool for producing TCRPNs. Firstly, we managed to design abstractly the tool, clarify its properties, and determine its

functionalities. Then it was needed to think mindfully the tool's classes and data structures. Lastly we implemented the code and debugged it.

1.3 Thesis Organisation

In our thesis we start by presenting some background information that is strongly associated with the topic and the formal definition of RPNs and their forms of reversibility. These are all included in Chapter 2.

In Chapter 3 we introduce the TCRPNs by defining them and their forms of reversibility.

We then proceed to Chapter 4 where we present our simulator for TCRPNs written in Clingo. Beside the presentation of the simulator, we also include some optimizations that were vital for reaching a higher efficiency.

In Chapter 5 we introduce our tool for generating TCRPNs and with it we mention some of its capabilities and some of its weaknesses.

Lastly, we conclude by summarizing our thesis and by addressing some challenges relevant to our work on the thesis, that we faced throughout the year. Additionally, we present our future goals associated with this thesis' topic.

At the end there is an Appendix that contains the GitHub repository that contains our implementation of the simulator and the tool for generating TCRPNs and the command line arguments we give to the simulator and to the tool that generates TCRPNs.

Chapter 2 Related Work

2.1 Introduction to Petri Nets 5

2.1.1 General Introduction 5

2.1.2 Formal Definition and Behaviour 5

2.1.3 Example of Application: The Candy Machine 6

2.2 Reversible Computation 8

2.3. Reversible Modelling 8

2.3.1 Forms of Reversibility 8

2.4 Reversing Petri Nets 9

2.4.1. Definition of Reversing Petri Nets 10

2.4.1. Forward Execution 13

2.4.3 Backtracking 14

2.4.4 Causal-Order Reversibility 15

2.5 Answer Set Programming 17

2.6 Clingo 17

2.6.1 Terms 18

2.6.2 Facts, Rules, and Constraints 18

2.1 Introduction to Petri Nets

2.1.1 General Introduction

Petri nets are a mathematical and graphical language applicable mainly to distributed and concurrent systems which can be used for the analysis of discrete event systems. They were first introduced in the 1960s by the German mathematician and computer scientist, Carl Adam Petri [3]. Petri nets are also applied in various systems, such as concurrent programming, and computational biology.

2.1.2 Formal Definition and Behaviour

The formal definition of a Petri Net is as follows [4].

Definition 1. A Petri Net is a five tuple (P, T, F, W, M_0) such that:

- P is a finite set of places,
- T is a finite set of transitions, with $P \cap T \neq \emptyset$,
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs,
- $W: F \to \{1,2, \dots\}$ is a weight function,
- $M_0: P \to \{0,1,2, \dots\}$ is the initial marking.

A Petri net is drawn as a bipartite directed graph. Each Petri net consists of two kinds of vertices, the places (represented graphically by circles) and the transitions (represented graphically by rectangles). On each place, there might exist some tokens (represented graphically by black dots). Furthermore, a Petri Net contains arcs that run between a place and a transition, and vice versa.

An instance of the distribution of the tokens, over the places of a Petri net, is called a marking. The initial marking corresponds to the initial distribution of the tokens over the places of a Petri Net.

Moreover, it holds that $W(p,t)$ is the weight of the arc from $p \in P$ to $t \in T$ and $W(t,p)$ is the weight of the arc from $t \in T$ to $p \in P$. If an arc (p,t) exists, where $p \in P$ and $t \in T$, then we say that p is an *input/incoming place* of t, and (p,t), is an *incoming arc* of t.

And if an arc (t, p) exists, where $p \in P$ and $t \in T$, then we say that p is an *output/outcoming place* of t, and (p, t), is an *output arc* of t.

We are now going to introduce some important concepts that describe the behaviour and execution of Petri nets.

- A transition t is said to be *enabled* if each input place p, of t, contains at least $W(p, t)$ tokens.
- A transition can *fire* only when it is enabled.
- A firing of a transition t, removes $W(p, t)$ tokens from each input place of t, p, and adds $W(t, p')$ tokens to each output place of t, p', where $t \in T$ and $p, p' \in P$.

2.1.3 Example of Application: The Candy Machine

In this section we present a scenario in which Petri Nets are applied

The scenario we are introducing is the modelling of a state machine. Specifically, the Petri Net depicted in Figure 2.1 found in [5], models a candy machine. This candy machine accepts only nickels (five cent coins) and dimes (ten cent coins) and does not return any change. It sells 15 and 20 cent candies.

In our example, the leftmost place, '0 cents' has two arcs starting from it and ending in two different transitions. Transition 'pay 5 cents' and transition 'deposit 10 cents' cannot be fired at the same time, instead, one of the two must be chosen each time.

The places of the Petri Net are 0 cents (p_0), 5 cents (p_1), 10 cents (p_2), 15 cents (p_3) and 20 cents (p_4), and the transitions are the rectangles named by their input conditions (incoming arcs weights) such as "pay 5 cents". In other words, it needs five tokens to fire. Each nickel is represented by five tokens and each dime by ten tokens.

Let us say for example that the initial state is marked by 5 tokens in place p_0. That means that the transition 'pay five cents', connected to p_0 by an arc, is enabled, since it only needs five tokens. It should be noted that the transition 'pay ten cents', connected to p_0, cannot be enabled because it needs ten tokens and not only five.

Let us assume now that the transition ‘pay five’ has fired. Thus, according to Petri Net’s behaviour all the five tokens will be transferred to p_1 and the marking is as it is seen in Figure 2.2

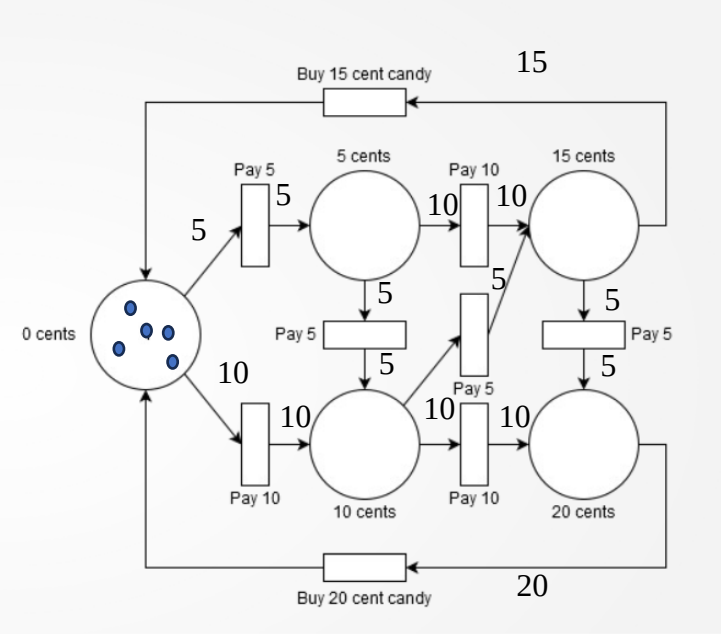

Figure 2.1 A Petri Net modelling a candy machine.

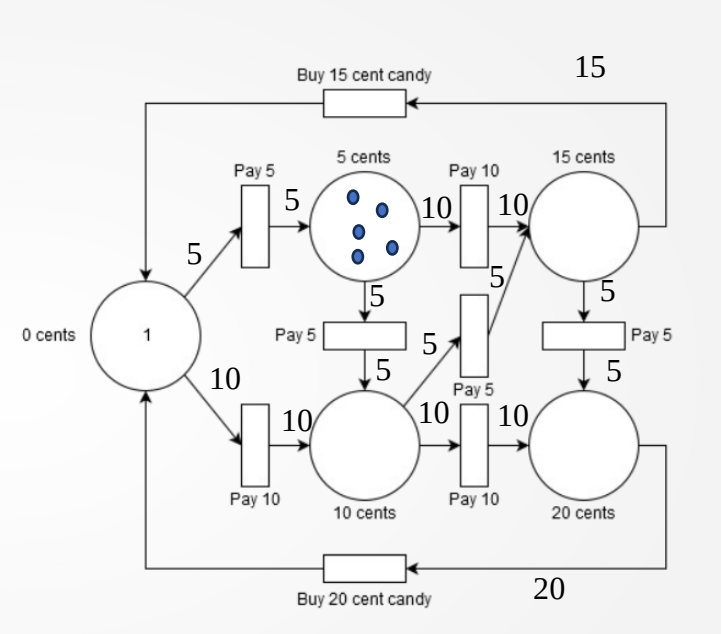

Figure 2.2 The Petri Net modelling a candy machine after the fire of the leftmost transition, ‘pay 5 cents’

2.2 Reversible Computation

Reversible computation is a form of computation that uses operations that can be easily and exactly reversed or undone [1]. The concept of reversible computation in science appeared the 1960's, when Rolf Landauer published a paper titled "Irreversibility and Heat Generation in the Computing Process". In it, it is stated that an irreversible operation that manipulates information stored in a computer, dissipates a minimum amount of heat [6].

Most laws of physics describing a closed system are reversible. In other words, previous states, of any previous time, are accessible. The reversible computation also applies in quantum-scale physics laws. However, as reported by Landauer, in non-frictionless environments, most operations are irreversible, and hence heat is dissipated. For example, a computer releases the overwritten data in the form of random information (entropy), that later becomes heat, instead of physically erasing it. This heat is energy, energy produced in every clock cycle, in which every active gate overwrites its previous output. [7]. Nowadays scientists are conducting research that aim to reverse gates' operations to decrease computers' heat dissipation.

2.3. Reversible Modelling

When modelling a system that supports reversible operations, difficulties arise.

Normally the system models do not offer machinery to remember their past states. Also, a vital question that arises is, what technique to use when going backwards. Three of the most powerful techniques are *backtracking, causal-order reversibility* and *out-of-causal reversibility.*

2.3.1 Forms of Reversibility

We begin our presentation of reversibility forms, with *backtracking*. Backtracking ensures that every state can be recursively undone, and the exactly previous state of the

reversed state can be reached. To put it in another way, we can only reverse that last operation that was executed.

The second form of reversibility is *causal-order reversibility*. Here, states can be reversed in an arbitrary order, if only they respect the causal dependencies. Any state can be reversed if caused actions are always undone before the actions that caused them [1].

For this form to be better understood, let us think of two operations, independent from each other, put differently, deprived of any causal dependencies between them. The first to be executed is op1 and the second, op2. Let us assume that both operations are to be reversed. Since the reversal is done in a causal-order fashion both transitions can be reversed in an arbitrary order, for example it is completely proper to reverse first op1 and then op2 or vice versa. Though, if the reversal was done in a backtracking fashion, op2 was obligated to be reversed before op1, since op2 was the last of them two to be executed.

The last form of reversibility is the *out-of-causal reversibility*. Out-of-causal reversibility, unlike backtracking and causal-order reversibility, ignores the causal dependencies completely. In operations that use backtracking and causal-order reversibility, we can move only back and forth to states that were previously accessed. The ability of out-of-causal reversibility is that operations can be reversed in any order and, when operations are reversed, new and unvisited states can be reached. States that were unreachable by forward execution, backtracking and causal-order execution.

The first two forms of reversibility will be discussed extensively and formally in the Chapter 2.4.

2.4 Reversing Petri Nets

Reversing Petri Nets (RPNs) are an extension of the traditional Petri nets that allow the reversal of the transitions. RPNs can simulate distributed systems and reach systems' past states to correct failures. RPNs are also applicable to biochemistry since they can model the process of catalysis.

Before introducing the RPNs formally, it is necessary to mention some decisions that were made to overcome a few obstacles that were found when we transitioned to reversing Petri Nets, whose transitions could go in both forward and backward directions.

The first design choice that was made, is the distinction between tokens. Each token must be considered as a unique entity.

Now, to capture the evolution of the tokens as well as the effects of a transition, we establish two new notions. The notion of the *base* and the notion of the *bond*. On the one hand, a *base/token* is an entity that neither can be consumed nor lose its individuality throughout the execution. On the other hand, a *bond* is the result of a transition. Bonds happen after the coalition of two bases/tokens. The tokens/bonds needed for a transition to fire are labelled on the incoming arc of a transition and the effect of a transition is labelled on the outcoming arc of a transition. Thus, whenever a transition is provided by the incoming places with the needed tokens and the transition fires, if it is indicated by the outcoming arcs that a new bond must be formed, the proper incoming tokens are attached with each other and form the bond. The result of this transition can be reversed and therefore the bond formed by the transition, breaks, and the tokens that composed the bond go back to their input places.

In succession to the previous decision, we now present the final design strategy that was chosen. With a view to monitor the history of the executions, and respectively to distinguish the predecessor of each transition, we present a new structure, the *history structure* that assigns a memory to each transition [1].

In this Chapter we give the formal definition of cyclic RPNs and of two forms of reversibility as presented in [1].

2.4.1. Definition of Reversing Petri Nets

Definition 2. A *reversing Petri net* is a tuple (*A, P, B, T, F*) where:

A is a finite set of *bases/tokens* ranged over by *a, b, …,*

P is a finite set of *places*,

$B \subseteq \{(a,b) | a,b \in A, and\ a \neq b\}$ is a set of undirected bonds ranged over by $\beta, \gamma, \ldots$. A bond $(a, b) \in B$ can be written as a-b, where a-b = b-a,

T is a finite set of transitions,

$F: (P \times T \cup T \times P) \rightarrow 2^{A \cup B}$ defines a set of directed *arcs* each associated with a subset of $A \cup B$, where, whenever $(a,b) \in F(x,y)$, then $a, b \in F(x,y)$.

The arcs are labelled with the names of the tokens/bonds needed to pass through them, either when the arcs are incoming to a transition, or outcoming. Hence if a label $l = F(x,y) = \{a\}$, then the token a must pass across the arc (x, y). As mentioned earlier, the labels of all the incoming arcs of a transition, designate all the tokens and bonds needed for a transition to be enabled. And all the labels of all the outcoming arcs of a transition are the resulting tokens or bonds, i.e. the effect of the transition. Assume that if a bond $(a,b) \in B, \beta \in F(x,y)$, then $a \in F(x,y)$ and $b \in F(x,y)$.

It needs to be stated that if $F(x,y) = \emptyset$, then there is no arc between x and y. As the previous notation was noted, we are now able to introduce two new notations. The first one is $\circ t = \{x \in P \mid F(x,t) \neq \emptyset\}$ and the second $t \circ = \{x \in P \mid F(t,x) \neq \emptyset\}$, for the incoming and outcoming places of transition t, respectively. Another notation required is $pre(t) = \cup_{x \in P} F(x,t)$ and $post(t) = \cup_{x \in P} F(t,x)$ for the union of all tokens and bonds labelled in the incoming and outcoming arcs of the transition t, respectively.

Now, it is crucial to point out that in the RPNs supported by this thesis, cycles are allowed. I.e. an incoming place of a transition can at the same time be an outcoming.

For each transition of the RPN we employ a history. The history, marked as $H: T \rightarrow 2^{\mathbb{N}}$, whenever returns an empty set for a transition, i.e. $H(t) = \emptyset$, it designates that the transition has not taken place and whenever $H(t) = \{k_1, k_2, \ldots, k_n\}$, where k_i is the i^{th} non-reversed instance of execution of the transition t, it means that the transition t was

not reversed for *n* times. This symbolism can be proved to be vital when modelling cycles in a Petri net. Lastly, we use $< M, H >$ to denote a state.

Based on the formal definition of RPNs, we now define the concept of *well* formedness that must hold during and after the construction of the RPN.

Based on the formal definition of RPNs, we now define the concept of *well* formedness that must hold during and after the construction of an RPN.

Definition 3. A reversing Petri net (*A, P, B, T, F)*, is *well formed* when the following hold $\forall t \in T$:

1) $A \cap pre(t) = A \cap post(t)$
2) If $a - b \in pre(t)$ then $a - b \in post(t)$
3) $F(t, x) \cap F(t, y) = \emptyset \ \ \forall x, y \in P, x \neq y$

These rules are explained below.

1) A transition cannot erase or create any tokens.
2) If a bond is a prerequisite for a transition to fire, then that bond cannot be destroyed by the transition. Instead, it must exist in an output place too.
3) Tokens and bonds cannot be cloned/duplicated to more than one outcoming arc of a transition.

In our machinery we also include an additional constraint associated with the initial state. It is constrained that; a token in an RPN initially exists in only one place, and the initial history must be empty for all the transitions.

Formally, for the initial state $< M_0, H_0 >$, it must be true that $|\{x | a \in M_0(x)\}| = 1, \forall a \in A$ and $H_0(t) = \emptyset, \ \forall t \in T$.

In our machinery we add the concept of connections as seen in the next definition.

Definition 4. $con(a, C) = (\{a\} \cap C) \cup$
$\{x | \exists w \ s.t. \ path(a, w, C), (b, c) \in w, x \in \{(b, c), b, c\}\}$

where, $path(a, w, C)$ if $w =< \beta_1, \beta_2, \dots, \beta_n >$, and $\forall\ 1 \leq i \leq n, \beta_i = (a_{i-1}, a_i) \in C \cap B, a_i \in C \cap A, and\ a_0 = a$.

2.4.1. Forward Execution

Definition 5. Consider an RPN (*A*, *P*, *B*, *T*, *F*), a transition $t \in T$ and a state <*M*, *H*>. Then *t* is *forward enabled* in <*M*, *H*> if:

1) If $a \in F(x, t), x \in P$ then $a \in M(x)$
2) If $\beta \in F(x, t), x \in P$ then $\beta \in M(x)$
3) If $a \in F(t, y_1), b \in F(t, y_2), y_1 \neq y_2$, then $b \notin con\big(a, M(x)\big)\ \forall\ x \in\circ\ t$
4) If $\beta \in F(t, x), \beta \in M(y)$ for some $y \in\circ\ t$, then $\beta \in F(y, t)$

In accordance with the previous definition, for a transition *t* to be forward enabled:

1) all the required tokens (tokens in $pre(t)$) must be available in an incoming place of *t*.
2) all the required bonds (bonds in $pre(t)$) must be available in an incoming place of *t*.
3) if two tokens are labelled on a different outcoming arc of *t*, then these tokens must not be connected to each other at any incoming place of *t*.
4) if an outcoming arc of t has a bond labelled on it and this bond is available on an incoming place of *t*, then this bond must be labelled on an incoming arc of *t* too.

Fresh bonds are the bonds formed by a transition. These bonds are the called the *effect* of a transition *t*, denoted by $eff(t) = post(t) - pre(t)$.

The definition that follows describes the effects of a transition that was executed in a forward fashion.

Definition 6. Consider an RPN (A, P, B, T, F), a state $< M, H >$, and an enabled transition t in the same state, we write $< M, H > \overset{t}{\rightarrow} < M', H' >$ where t is an enabled transition:

$$M'(x) = \begin{cases} M(x) - \bigcup_{a \in F(x,t)} con(a, M(x)) & if\ x \in \circ t \\ M(x) \cup F(t,x) \cup \bigcup_{y \in P, a \in F(t,x) \cap F(y,x)} con(a, M(y)) & if\ x \in t \circ \\ M(x), & otherwise \end{cases}$$

$$H'(t') = \begin{cases} H(t') \cup \{\max(\{0\} \cup \{k \mid k \in H(t''), t'' \in T\}) + 1\}, & if\ t' = t \\ H(t') & otherwise \end{cases}$$

The effects of the forward execution of a transition are shown in the above definition.

1) All tokens/bonds needed for the transition to fire are moved with their connected components from the incoming places of the executed transition, to the outcoming. The tokens/bonds that were neither prerequisite for a transition to fire, nor connected to tokens that were, are not removed from their places.
2) In the history list of t we append, the previous largest history instance of all transitions, plus one (or just one if none of the transitions was executed before). The history of the non-executed transitions remains the same as it was in the previous state before the execution of t.

2.4.3 Backtracking

Definition 7. Consider a state $< M, H >$ and $t \in T$. We say that t is *bt-enabled* in $< M, H >$ if:

1) $\forall x \in t \circ$, if $a \in F(t,x)$, then $a \in M(x)$ and if $b \in F(t,x)$, then $b \in M(x)$ and
2) $\exists k \in H(t)$ with $k' \leq k\ \forall k' \in H(t'), t' \in T$.

A transition t can be backtracked if:

1) It is the last transition to be forwardly executed (has the highest history instance among all transitions).
2) all the tokens/bonds that are labelled on the outcoming arcs of t, must be placed in t's outcoming places.

The subsequent definition describes the effects of a transition that was backtracked.

Definition 8. Consider an RPN (A, P, B, T, F) a state $< M, H >$, and a transition t, that is *bt-enabled* in $< M, H >$. We write $< M.H > \overset{t}{\rightsquigarrow_b} < M', H' >$ where t is a bt-enabled transition in $< M, H >$ and:

$$M'(x) = \begin{cases} M(x) \cup \bigcup_{y \in t\circ, a \in F(x,t) \cap F(t,y)} con\big(a, M(y) - eff(t)\big) & if\ x \in \circ\, t \\ M(x) - \bigcup_{a \in F(t,x)} con\big(a, M(x)\big) & if\ x \in t \circ \\ M(x), & otherwise \end{cases}$$

$$H'(t') = \begin{cases} H(t') - \{k\} & if t' = t, k = \max\,(H(t)) \\ H(t') & otherwise \end{cases}$$

Informally, when an execution of t is backtracked,

1) all the tokens labelled on the outcoming arcs of t, are removed from t's outcoming places and are relocated to t''s proper (as designated by the incoming arcs of t) incoming places.
2) any fresh/new bonds created by t, are destroyed.
3) the largest history instance of t's history is deleted.

2.4.4 Causal-Order Reversibility

We first define the term *transition occurrence.*

Definition 9. Consider a state $< M, H >$ and a transition t. We refer to (t, k) as *transition occurrence* in $< M, H >$ if $k \in H(t)$.

Definition 10. Now, consider a state $< M, H >$ and suppose $< M, H > \overset{t}{\rightarrow} < M', H' >$ with $(t, k), (t', k')$ transition occurrences in $< M', H' >$, $k = \max\,(H'(t))$. We say that (t, k) *causally depends* on (t', k') denoted by $(t', k') \prec (t, k)$, if $k' < k$ and there exists $a \in F(x, t)$ where $con\big(a, M(x)\big) \cap post(t') \neq \emptyset$.

So, by definition, in $< M', H' >$, the occurrence (t, k) causally depends on the occurrence(t', k'), if t was executed after t', and t to fire requires one or more tokens/bonds produced by t'.

Proceeding, we are going to define exactly when a transition can be reversed in a causal order manner and what arises after the reversal. The definitions need an additional piece in our machinery, which is a set of all the causal dependencies arose up to an RPN's state. Therefore, we extent the parameters of a state by adding this set, denoted by $\prec$. And now a state is represented by the triple $< M, H, \prec >$. Initially, $\prec_0 = \emptyset$.

Definition 11. Given a reversing Petri net (A, P, B, T, F), a state $< M, H, \prec >$ and a transition t forward-enabled in $< M, H >$, we write $< M.H. \prec > \xrightarrow{t} < M', H', \prec' >$ where M' and H' are defined as in definition of the effects of forward execution and

$\prec' = \prec \cap \{((t', k'), (t, k)) | k = \max(H'(t)), (t, k)$ causally depends on $(t', k')\}$

Definition 12. Given a reversing Petri net (A, P, B, T, F), a state $< M, H, \prec >$ and a transition $t \in T$. Then $t, H(t) \neq \emptyset$, is *co-enabled* (causal-order enabled) in $< M, H, \prec >$ if:

1. $\forall x \in t \circ$, if $a \in F(t, x)$ then $a \in M(x)$ and if $\beta \in F(t, x)$, then $\beta \in M(x)$ and
2. $there\ is\ no\ transition\ occurence\ (t', k') \in < M.H, \prec >$ with $(t, k) \prec (t', k')$, for $k = \max(H(t))$.

In other words, a transition t is co-enabled if:

1) all the outcoming places of t hold all the required tokens/bonds, for the transition to reversibly fire.
2) there is no occurrence (t', k') that causally depends on the last execution of t. This condition is more relevant in the presence of cycles.

Definition 13. Now, given an RPN (A, P, B, T, F) and a co-enabled transition t in $< M, H, \prec >$, we write $< M.H. \prec > \xrightarrow[c]{t} < M', H', \prec' >$, for M' and H' as in definition of the effects of backtracking and $\prec'$ such that

$\prec' = \{\big((t_1, k_1), (t_2, k_2)\big) \in \prec \mid (t_2 \neq \max\big(H(t)\big)) \}$.

Consequently, as described by the previous definition, the effects of causal-order reversibility are the same as the effects of backtracking with a small addition. All references to the reversed transition occurrences, are removed from $\prec'$.

2.5 Answer Set Programming

Answer Set Programming (ASP) is a form of declarative programming that focuses on NP-hard search problems. It is based on the *stable model (answer set)* semantics of logic programming. In ASP, search problems are reduced to computing stable models [8]. That is, by computing answer sets, the search problem is also solved. Models are generated by the *answer set solver*.

It is necessary to note that most ASP Solvers perform their search procedure by using an enhanced DPLL algorithm, meaning that the solvers always terminate (unlike Prolog query evaluation).

How ASP works is analysed in the next subchapter, where we discuss an ASP language, Clingo.

2.6 Clingo

Clingo is a declarative ASP language, developed at the University of Potsdam.

Clingo couples a grounder, *Gringo* and a solver named *Clasp*. The grounder receives from the user a logic program consisting of variables and then translates it to an equivalent ground (without variables) program, which is given to the solver. The solver is responsible for producing the answer sets, which are the output of Clingo.

In the subchapter that follow, we present briefly the syntax and semantics of the input language of Clingo.

2.6.1 Terms

ASP programs include *terms*. The terms are mostly used as arguments of atoms (discussed in the next subsection). The basic building blocks are the *simpleterms (simple terms)*. Simple terms of a program are the *strings*, the *integers*, the *variables* (written with capital case letters), and the *constants* (written with lowercase letters). Beside the basic block terms there are also *tuples* and *functions* [9].

2.6.2 Facts, Rules, and Constraints

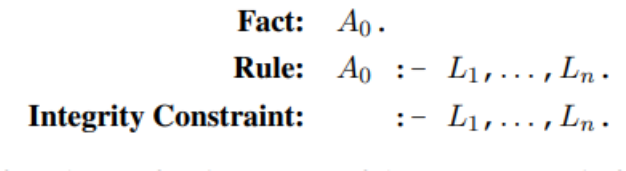

Figure 2.3: Facts, rules and constraints

Without any significant exceptions, every ASP language, and therefore Clingo, includes *rules*. Each rule is a construct that consists of a *head* and a *body*. The head of the rule in Figure 2.3 is the atom A_0. *Atoms* are the expressions that contain a predicate symbol, followed by a parenthesis that contains terms. The body of the same rule is the disjunction of a finite number of literals ($L_1, \ldots, L_n$). *Literals* are atoms or the negative form of atoms (atoms preceded by *not)*. The symbol '*:-'* can be translated as the word '*if'* and equally as the logical operator '←' (implication). Thus, the entire rule is translated as 'if the disjunction of the literals $L_1, \ldots, L_n$, has the value true, then the atom A_0 is true'. When the atom of the head of a rule is inside { }, then the rule is a *choice rule*. Choice rules describe all the alternate ways to form a stable model. An atom of a choice rule in a model can be satisfied and, in another model, of the same program, cannot be satisfied.

Facts are the rules without a body, so they are always true.

Constraints are the rules that filter solution candidates, meaning that the literals in their body must not all together be true.

Chapter 3 True Concurrent Reversing Petri Nets

3.1 Formal Definition 20

3.2 Forward Execution 20

3.3 Backtracking 21

3.4 Causal-Order Execution 22

True Concurrent Reversing Petri Nets (TCRPNs) are an extension of cyclic Reversing Petri nets, with the additional capability of firing transitions or reversing executions of transitions, with true concurrency.

In the context of Petri Nets and related computational models, true concurrency means that transitions can fire independently and at the same time. This means that the execution of one transition does not interfere with the execution of another transition occurring simultaneously. This ensures that multiple transitions can fire concurrently without affecting each other's execution.

In a Reversing Petri net, when multiple transitions are enabled, only one of the enabled transitions can fire at any given time. The remaining transitions must wait for their turn to fire in subsequent steps.

In large-scale systems, the overhead of serializing the execution of transitions to reach a goal state can be prohibitive. Since TCRPNs can fire transitions concurrently, they aim to achieve higher efficiency and reduce the overhead associated with the wait times that RPNs experience.

3.1 Formal Definition

The formal definition of True Concurrent Reversing Petri Nets is the same definition as the definition of RPNs (Definition 2), given in Chapter 2.4.1.

TCRPNs are well-formed, that is, the Definition 3 applies to all TRCPNs. Furthermore, for the initial state of a TCRPN $< M_0, H_0 >$, similarly with RPNs, all the tokens must be placed at only one place of the TCRPN and the initial histories of all the transitions are empty.

3.2 Forward Execution

The definition of forward enabledness in RPNs applies to TCRPNs too. So, if the conditions outlined in Definition 5 for forward enabledness in RPNs are met for a transition in a TCRPN, then that transition is also considered enabled in the TCRPN. This suggests that the criteria for forward enabledness are transferrable from RPNs to TCRPNs.

However, there is a difference between the definition of forward execution in TCRPNs, than in RPNs, since they allow concurrent execution of transitions.

The definition of forward execution is as follows.

Definition 14. Consider an TCRPN (A, P, B, T, F), a state $< M, H >$, and a set of all the enabled transitions ET in the same state, we write $< M, H > \xrightarrow{ET' \subseteq ET} < M', H' >$ where:

$$M'(x) = \begin{cases} M(x) - \bigcup_{a \in F(x,t)} con\big(a, M(x)\big) & if\ \exists t \in ET' : x \in \circ\, t \\ M(x) \cup F(t,x) \cup \bigcup_{y \in P, a \in F(t,x) \cap F(y,x)} con\big(a, M(y)\big) & if\ \exists t \in ET' : x \in t \circ \\ M(x), & otherwise \end{cases}$$

$$H'(t') = \begin{cases} H(t') \cup \{\max(\{0\} \cup \{k \mid k \in H(t''), t'' \in T\}) + 1\}, & if\ t' \in ET' \\ H(t') & otherwise \end{cases}$$

And

For $ET' \subseteq ET$ it holds that:

1) ET' is an arbitrary subset of ET
2) $\forall\, t_1, t_2 \in ET'$where $t1 \neq t2, if\ a \in F(x, t_1)$ then $con\big(a, M(x)\big) \cap F(x, t_2) = \emptyset$ $\forall x \in P$

In accordance with the Definition 14**,** at a state $< M, H >$ of a TCRPN, an arbitrary set of enabled transitions fired concurrently and created the state $< M', H' >$.

Nonetheless, it must hold that if two transitions fire at the same time, then the required tokens of the one transition must not be connected with any of the prerequisite tokens of the other transition.

3.3 Backtracking

The requirements for a transition of a TCRPN to be bt-enabled are the same requirements stated in the Definition 7 for a transition of an RPN to be bt-enabled.

Definition 15: Consider an TCRPN (A, P, B, T, F) a state $< M, H >$. We write $< M.H >$ $\overset{BET' \subseteq BET}{\rightsquigarrow}_b$ $< M', H' >$ where BET are all the bt-enabled transitions in $< M, H >$ and:

$$M'(x) = \begin{cases} M(x) \cup \bigcup_{y \in t\circ, a \in F(x,t) \cap F(t,y)} con\big(a, M(y) - eff(t)\big) & if\ \exists t \in BET' \colon x \in \circ\, t \\ M(x) - \bigcup_{a \in F(t,x)} con\big(a, M(x)\big) & if\ \exists t \in BET' \colon x \in t \circ \\ M(x), & otherwise \end{cases}$$

$$H'(t') = \begin{cases} H(t') - \{k\} & if\ t' \in BET', k = \max\,(H(t')) \\ H(t') & otherwise \end{cases}$$

And BET' is an arbitrary subset of BET.

In Definition 15 it is noticeable that without exceptions, all the bt-enabled transitions can fire. In backtracking there is no possibility for two bt-enabled transitions to require same tokens otherwise that would mean that the two transitions fired in previous timesteps together, even though the needed same resources. In the following definition, of causal-order execution, we see that the same observation holds true for causal-order execution too.

3.4 Causal-Order Execution

Definition 9 is relevant in the context of TCRPNs as well.

Definition 16. Consider a state $< M, H >$ of a TCRPN and suppose $< M, H > \overset{ET' \subseteq ET}{\underset{\rightarrow}{}} < M', H' >$ with $(t, k), (t', k')$ transition occurrences in $< M', H' >, k = \max\big(H'(t)\big), \forall t \in ET'$. We say that $\forall t \in ET'$ (t, k) *causally depends* on (t', k') denoted by $(t', k') \prec (t, k)$, if $k' < k$ and there exists $a \in F(x, t)$ where $con\big(a, M(x)\big) \cap post(t') \neq \emptyset$.

Definition 17. Given a TCRPN (A, P, B, T, F), a state $< M, H, \prec >$. We write $< M. H. \prec > \overset{ET' \subseteq ET}{\underset{\rightarrow}{}} < M', H', \prec' >$ where M' and H' are defined as in definition of the effects of forward execution, ET contains all the enabled transitions, for ET' it holds everything that holds for ET' in Definition 14 and

$\prec' = \prec \cap \{\big((t', k'), (t, k)\big) \big| \forall t \in ET', k = \max\big(H'(t)\big), (t, k)$ causally depends on $(t', k')\}$

The definition of causal-order enabledness, **Definition 12,** applies to TCRPNs too.

Definition 19. Now, given a TCRPN (A, P, B, T, F) and a set COET with all the transitions co-enabled in $< M, H, <>$, we write $< M. H. <> \overset{COET' \subseteq COET}{\underset{c}{\twoheadrightarrow}} < M', H', <'>$, for M' and H' as in definition of the effects of backtracking and $\prec'$ such that

$$\prec' = \{\big((t_1, k_1), (t_2, k_2)\big) \in \prec | (t2 = t \rightarrow k_2 \neq \max\big(H(t)\big)) \ \forall t \in COET'\}.$$

Where $COET'$ is an arbitrary subset of $COET$.

Chapter 4 – TCRPNs Simulator

4.1 Simulating a TCRPN 24

4.2 Simulating Forward Execution 26

4.3 Simulating Reverse Execution 27

4.4 Optimizations 28

4.5 Weaknesses 29

In this chapter of the thesis, we are about to simulate a TCRPN in ASP. The code of the program can be found in this thesis' GitHub repository written in Appendix in the directory src.

The language chosen for developing the simulator is Clingo. Clingo, is well-suited for this purpose due to its robust features for solving complex search problems, such as exploring TCRPNs' reachable states.

Our simulator runs for a specified number of timesteps (rounds) and returns 'Satisfiable' if it finds at least a path that passes through a goal state. Each model generated by the simulator represents a different path of reachable states, ensuring that all goal states are reached in each path.

4.1 Simulating a TCRPN

The input of the simulator is a TCRPN, one or more goal states and the maximum number of timesteps.

The output of the simulator are the models with the execution path that satisfy the goal state. Or the word 'UNSATISFIEABLE' if the goal states cannot be reached.

As specified by the definition of TCRPNs, a TCRPN is composed of places, transitions, arcs, tokens, and bonds.

Before proceeding to the encoding of the TRPN in ASP, we should note that in our implementation we assumed that if $(a, b) \in B$ and $(a, b) \in F(x, y)$, then $a, b \notin F(x, y)$ and furthermore if $(a, b) \in B$ and $(a, b) \in M(x)$, then $a, b \notin M(x)$. These assumptions were made to limit the possible predicates and distinct the tokens that are not part of a bond with the tokens that are part of a bond.

We now proceed to construct a TCRPN and a simple goal state for a timestep *ts* with the following facts.

- Tokens: `token(a).`
- Transitions: `trans(t).`
- Places: `place(p).`
- Incoming arcs: `incoming(p,t,a).incomingbond(p,t,a1,a2).`
- Outcoming arcs: `outcoming(t,p,a). outcomingbond(t,p,a1,a2).`
- Initial placement: `placeholds(p,a,0). placeholdsbond(p,a1,a2,0).`
- Goal state: `placeholds(p,a,ts). placeholdsbond(p,a1,a2,ts).`

It is important to ensure that any constants for places, tokens, and transitions used in the predicates *incoming*, *outcoming*, *incomingbond*, *outcomingbond*, *placeholds*, *and placeholdsbond* must be also defined in the respective predicates' *places*, *tokens*, *and transitions*.

Before simulating it is crucial to examine if the TCRPN is well formed or not based on the Definition 3. If it is not well formed, it can be discarded. Simulator works correctly only for well-formed inputs, so this check is vital.

We must also check the initial state of the TCRPN, in which each token must appear once, in only one place. We do not check if the initial state is 0, because the program cannot tell the difference between the initial placements than the goal placements.

The implementation can be seen clearly in the files src/well_formed.Clingo and src/initial_state.Clingo respectively. Both programs' input is a TCRPN as constructed above. Note that these two programs are not part of the simulation.

4.2 Simulating Forward Execution

The program that models forward execution is used by both the program that models backtracking and the program that models causal-order execution.

We note that in forward execution and in reverse execution if a bond (a, b) exists then the bond (b, a) does not exist, even though the two bonds are the same. This is done for optimization. However, in the program that checks for well-formedness and the program that checks the initial state, every predicate that holds for a bond (a, b) holds for a bond (b, a). We did not optimize these two programs because they are not part of the simulation. We preferred to keep the simple.

What happens at each timestep in each model:

- The simulator finds all the enabled transitions of the given TCRPN. To find all the enabled transitions, the simulator discards all the transitions that do not satisfy the requirements of the Definition 5 and enables the remaining transitions.
- Some (or none) enabled transitions are fired. In accordance with the Definition 14, we add constraints to disallow the concurrent firing of two transitions that have the same prerequisites. Only one of them can be fired, or none.
- The effects of the transitions that fired in the previous timestep are produced (except from the initial timestep).

4.3 Simulating Reverse Execution

To encode backtracking and causal-order execution in True Concurrent Reversing Petri Nets, we adopt the same approach for both mechanisms.

It should be mentioned that in all the tests we have taken, the given TCRPNs used only one of the two forms of reversibility throughout their execution. That is for each input we would run either the programs src/backtracking.Clingo with src/forward_for_reversal.Clingo or the programs src/causal_execution.Clingo with src/forward_for_reversal.Clingo.

What happens at each timestep in each model:

- The simulator finds all the bt-enabled and co-enabled transitions of the given TCRPN. To find all the bt-enabled and co-enabled transitions, the simulator discards all the transitions that do not satisfy the requirements of the Definition 7 and the Definition 12 respectively and enables the remaining transitions.
- Some (or none) of the bt-enabled or co-enabled transitions are reversed, in accordance with Definitions 15 and 19, respectively. Some constraints must be added to manage certain cases that occur. More details are provided after the following point.
- The effects of the transitions that fired in the previous timestep are produced (except from the initial timestep).

It was mentioned in Chapter 3 that two bt-eanbled or co-enabled transitions never require same tokens/bonds at the same timestep. Nevertheless, there is a possibility that a transition is both bt-enabled, or co-enabled and at the same time forward enabled. This case can occur when a transition's incoming places are also its outcoming. For that reason, we add a constraint to prevent the transition to be executed both forwardly and backwardly at the same time.

There is also a possibility that the tokens in a place are needed by two different transitions at the same time, one being forward enabled and the other being bt-enabled or co-enabled. Again, we allow only one of them to execute by adding constraints.

4.4 Optimizations

In this chapter we present some of the techniques we implemented to optimize our simulator. The bottleneck in large scale TCRPNs is the grounder, so our optimizations are mostly grounder oriented.

The first optimization we implemented was when a bond (a,b) exists then the bond (b,a) does not exist even though the bonds are equivalent. That is all the predicates that hold for (a,b) do not hold for (b, a). The reason we implemented this optimization is to assist the grounder by reducing the predicates it needs to keep track of.

An additional optimization is to add “types” of variables. E.g. when an atom needs to ground more than one variable, we use supplementary atoms to designate the type of each one of the variables. By explicitly specifying the types of variables, you can potentially reduce the search space by guiding the grounding process. Clingo can optimize its grounding and solving algorithms based on the provided variable types, leading to faster solving times for complex problems.

Moreover, the last optimization technique we implemented is to discard the states that all the transitions do not perform an action. That is, we limit the number of possible states. With fewer states to consider, the overall execution time of the system is likely to decrease. This optimization helps in improving the efficiency of the system by minimizing unnecessary computation and exploration of non-productive states.

4.5 Weaknesses

We run the simulator for a number of iterations, and we observed that the execution time of the simulator, on some inputs, is increased dramatically. Specifically, we run the simulator 30 times for 15 timesteps each and each time the number of transitions increased. When we reached 43 transitions Clingo was not able to give any output.

However, these results are not surprising since grounding is an NP-hard problem. The ground atoms are increasing exponentially with the increment of transitions and the increment of timesteps.

Chapter 5– Tool for Generating TCRPNs

5.1 A Tool for Generating Petri Nets 29

5.2 Components 30

5.3 Bridges 31

5.4 Main TCRPN 31

5.5 Implementation Details 33

5.6 Capabilities 34

5.7 Weaknesses 34

Since we created a simulator that simulates TCRPNs, the next step is to develop a tool that generates a variety of TCRPNs to use as input for our simulator. This will allow us to observe TCRPNs' behaviour and assess their benefits and drawbacks.

We implemented our code in Python 3. Python 3 offers a wide variety of data structures, making it an ideal choice for our purpose. The code can be found in the GitHub repository mentioned in Appendix in the folder petri_net_producer.

5.1 A Tool for Generating Petri Nets

The tool we developed accepts as input a total number of transitions and a maximum degree, both are integers. For simplicity, in this Chapter we denote the total number of transitions as total_trans and the maximum degree as max_degree. Beside these two integers, the tool's input must contain several components and several bridges.

The components are small TCRPNs consisting of a small number of places and a small number of transitions. Bridges consist of a transition and connects compatible components with each other. We connect components with bridges to build a big TCRPN, the main TCRPN, as we call it.

The output of the tool is a TCRPN (the main) that contains exactly total_trans transitions and max_degree parallel sequences of components. We explain components, bridges, and sequences in the next subchapters.

5.2 Components

Components are small TCRPNs that are given as input to the tool. We characterize them as small because they are always smaller than the main TCRPN.

The tool receives at its starting point some of the tokens/bonds needed for the transitions to fire and exports from the ending point some tokens/bonds.

A component consists of:

- Places. Each place can be the starting point of a component, the ending point of component or just a middle point. Each component has only one starting point and only one ending point.
- Transitions. Each component contains a few transitions. The transitions of a component only require tokens that are initially placed in the component or needed tokens/bonds.
- Needed Tokens and Needed Bonds. Needed tokens and bonds are some of the tokens and bonds that the component's transitions require to fire.
- Tokens. Are the tokens that are initially placed inside a component at one of its places. A component initially cannot place bonds (for simplicity).
- Placements. The initial distribution of tokens in the places of the component.
- Out Tokens and Out Bonds. Out tokens and out bonds are the tokens and respectively bonds that the component exports.

- Arcs. Are the arcs that connect places with transitions. They can be incoming or outcoming or both. We must pay attention to the fact that the arcs of a component never connect places and transitions of the component to places and transitions outside the component. The are labelled with needed tokens or needed bonds or with tokens that the component contains initially.

The tokens the places and the transitions have a unique name.

5.3 Bridges

A component *A* whose union of out tokens and out bonds is a superset of a component *B*'s union of needed tokens and needed bonds, is *compatible* with *B*. And *A* is the *sender* and *B* the *receiver*.

Bridges link compatible components. A bridge contains a transition that connects to the endpoint of a sender with an incoming arc and to the starting point of a compatible receiver with an outgoing arc.

A bridge consists of:

- Start Component. Start component is a sender.
- End Component. End component is a compatible to sender, receiver.
- Transition.
- Arcs. The arcs incoming to the transition (those that start from the sender) and the arcs outgoing from the transition (those that end at the receiver).

5.4 Main TCRPN

The main TCRPN is a combination of compatible bridges and components. It contains total_trans transitions.

In addition to the components received from the user, the main TCRPN includes a supplementary component called the initial component. This initial component consists

of only one place and does not include any additional elements of those described in **Chapter 5.2**. The place in the initial component has a degree of max_degree, meaning it is connected to max_degree initial bridges and acts as the sender for each bridge.

Each initial bridge is dynamically created and connected to a random receiver from the input components, referred to as the initial sequence receivers. From these receivers, we start max_degree sequences. In each sequence, we add compatible bridges and components until the main TCRPN reaches a total of total_trans transitions. If a sequence runs out of compatible components but the total number of transitions is still less than total_trans, we add extra bridges until max_trans is reached that start from a random component within the sequence and end back at the initial place.

Since each component can appear in the main TCRPN multiple times, beside their unique name, we assign a unique identifier to each transition and place within the component. The unique identifier corresponds to the number of times each component or bridge is included in the main TCRPN.

Initially, all the tokens/bonds needed by the initial sequence receivers are placed in the initial component. There are enough unique tokens/bonds for all the initial receivers in each sequence, and each token/bond is identified by a name and the sequence number it is intended for.

Figure 5.1 illustrates an example of a main TCRPN with max_degree=2 and total_trans=4.

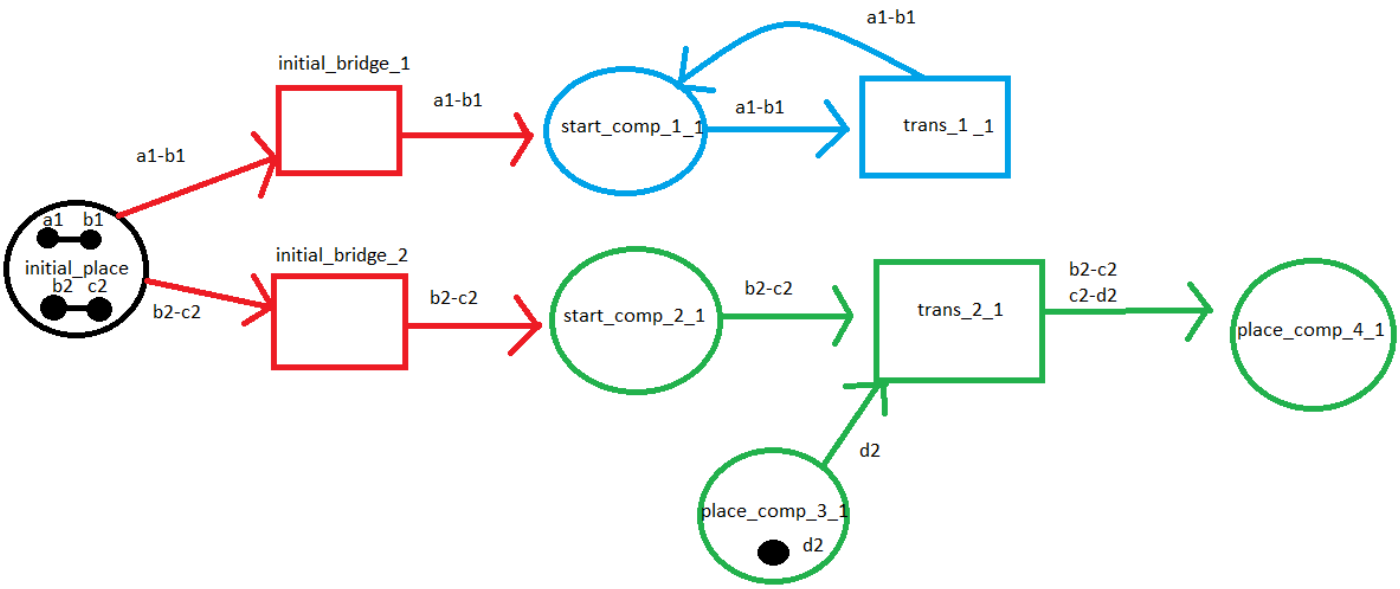

Figure 5.1 A main TCRPN

The black component is the initial component. The red bridges are the initial bridges. The blue and green components are the initial sequence receivers. The last integer after places and transitions' names is their frequency in the main TCRPN and the integer next to tokens' names is the number of the sequence they are intended for.

5.5 Implementation Details

We note that in our implementation we made some assumptions. The first assumption is that all the components and bridges, received by the user, are well formed and their initial state is correct based on the Definition 3 and the standards of the initial state**.**

Secondly, for the tool to work properly, we assume that at least one component can be added to each sequence. To ensure this, we require the user to maintain the ratio:

$total_{trans} \geq 2 \times max_{degree}$when providing input. Additionally, among the input components, there must be at least one component containing only one transition. This ensures that, in addition to the max_degree initial transitions of the initial bridges, we can add at least one component in each sequence.

5.6 Capabilities

Despite extensive searching, we couldn't find an existing tool for RPNs that met our specific requirements. Consequently, we developed a tool capable of generating TCRPNs and therefore RPNs, with customizable numbers of transitions and degrees.

Since the definition of RPNs is equivalent to the definition of TCRPNs, this tool is suitable for the construction of RPNs.

5.7 Weaknesses

While our tool can reliably generate TCRPNs meeting specified requirements without exceptions, it faces limitations when compatibility between components is constrained. In such cases where certain components cannot provide others, and dead ends are encountered in each sequence, the only recourse is to add extra bridges. However, this may result in less expressive and more complex output TCRPNs.

Additionally, our tool lacks the ability to scale TCRPNs based on a specific target number of places or tokens; it can only scale based on the number of transitions and maximum degree.

Chapter 6 Conclusion

6.1 Summary 35

6.2 Challenges 36

6.3 Future Work 36

6.1 Summary

In this thesis, we have explored the development and application of Concurrent and Reversible Petri Nets (TCRPNs), an innovative extension of traditional Reversing Petri Nets (RPNs). TCRPNs have the capability for true concurrency, allowing multiple transitions to fire simultaneously and independently. While we assume that this enhancement could improve the efficiency of simulating and analysing distributed systems due to the concurrent execution, further research is needed to validate this assumption.

To accurately capture and study the behaviour of TCRPNs, we employed Answer Set Programming (ASP). ASP's suitability for addressing NP-hard search problems made it an ideal choice for modelling the intricate dynamics of TCRPNs and exploring their reachable states. Our simulation approach demonstrated that ASP could effectively manage the complexity inherent in TCRPNs.

Recognizing the novelty of TCRPNs and the potential interest they could generate within the research community, we identified a critical need for a robust dataset of TCRPN models. To meet this need, we developed a specialized tool designed to generate a variety of TCRPNs. This tool supports ongoing research by providing necessary data.

In conclusion, our work lays the groundwork for future studies on TCRPNs, offering a new perspective on efficient system simulation. The development of our TCRPN generation tool represents a significant step towards enabling comprehensive research and practical applications of TCRPNs in the field of distributed systems.

6.2 Challenges

The journey I have undertaken during this year of working on my thesis dissertation has been both fulfilling and challenging. As this was my first time tackling such an extensive and important project, it required a significant amount of dedication and perseverance.

As expected, I encountered several challenges while working on my thesis. The first significant challenge was creating the formal definition of TCRPNs and their forms of reversibility. Although some parts of the definitions were like those of RPNs, devising the new definitions was not straightforward. TCRPNs are a complex system that requires a deep understanding of their properties.

One significant challenge we faced was the unpredictable difficulty in finding or creating a tool for producing TCRPNs. Despite extensive research over several months, we were unable to find an existing tool that met our specific requirements. This compelled us to develop a tool ourselves. The lack of existing solutions highlighted the novelty of our work but also added an unexpected layer of complexity to our project.

The development of the final tool was also complicated. Designing a tool that could reliably generate TCRPNs involved numerous technical challenges and iterations. Ensuring the tool's accuracy and efficiency required precise testing and refinement. This development phase not only tested our technical skills but also demanded a high level of problem-solving abilities.

6.3 Future Work

The goals associated with the topic of this thesis that we set for the future are numerous. Refactoring a tool for True Concurrent Reversing Petri Nets involves several key steps aimed at enhancing its efficiency and usability. Firstly, cleaning up the existing codebase is crucial to improve readability and organization. Simplifying complex sections and removing redundant code can significantly enhance the overall quality of the codebase. Clear and concise code facilitates easier maintenance. Abstraction is another essential

aspect of refactoring the tool. Identifying common functionalities and patterns within the codebase allows for the creation of reusable components or functions.

In the pursuit of advancing the tool, a crucial goal for the future is to observe and understand its limitations, paving the way for devising strategies to address them. This includes optimizing algorithms to improve performance and modifying fundamental features to enhance scalability, accommodating larger TCRPNs with spepcified numbers of places or tokens. By tackling these constraints directly, the tool can evolve into a more robust solution, better equipped to handle the complexities of modelling and analysing systems.

Finally, it is important that we recognize the potential that TCRPNs have in distributed and concurrent systems and do the research they deserve to determine not only their possible benefits but also their disadvantages and the overhead they might cause. Only in that way we will be able to utilize them in the best way possible.

Appendix

GitHub Repository 38

Command Line Input and Example of Execution of the Simulator 38

Command Line Input of Tool for Generating TCRPNs 40

GitHub Repository

The code of the simulator and the tool for producing TCRPNs is in the following repository in the directories src and petri_net_producer respectively.

https://github.com/lioann03/petri_net_modelling_tool.

Command Line Input and Example of Execution of the Simulator

To run the simulator, we need two Clingo files, the file src/forward_for_reversal.Clingo with either src/backtracking.Clingo or src/causal_execution.Clingo. In our example we chose the program src/backtracking.Clingo.

The terminal command is:

```
Clingo src/backtracking.Clingo src/forward_for_reversal.Clingo input1 -c ts=3 0
```

Where,

- `input1:` is the input file that contains the TCRPN and its goal state.
- `-c ts=3:` determines the maximum number of timesteps.
- `0:` forces Clingo to print all the computed models.

In our example the input file (input1) contains the following TCRPN

```
trans(t110).trans(t40).trans(inittrans0).trans(inittrans1).token(token30).token(token40)
.token(token50).

token(token60). token(token10). token(token20). token(token70). token(token80).
token(token31). token(token41).

token(token11). token(token21). token(token51). token(token61).

place(r00). place(z20). place(z10). place(z00). place(init_place).

placeholdsbond(init_place,token30,token40,0).
placeholdsbond(init_place,token50,token60,0).
placeholdsbond(init_place,token10,token20,0).
placeholdsbond(init_place,token70,token80,0).
placeholdsbond(init_place,token31,token41,0).
placeholdsbond(init_place,token11,token21,0). placeholds(z1,token51,0).
placeholds(z1,token61,0).

incomingbond(init_place,inittrans0,token10,token20).outcomingbond(inittrans0,r00,token50
,token60).incomingbond(init_place,inittrans0,token70,token80).outcomingbond(inittrans0,r
00,token70,token80).incomingbond(init_place,inittrans0,token30,token40).outcomingbond(in
ittrans0,r00,token10,token20).outcomingbond(inittrans0,r00,token30,token40).incomingbond
(init_place,inittrans0,token50,token60).incomingbond(init_place,inittrans1,token11,token
21).outcomingbond(inittrans1,z00,token31,token41).incomingbond(init_place,inittrans1,tok
en31,token41).outcomingbond(inittrans1,z00,token11,token21).outcomingbond(t110,r00,token
10,token20).incomingbond(r00,t110,token70,token80).outcomingbond(t110,r00,token30,token4
0).incomingbond(r00,t110,token10,token20).outcomingbond(t110,r00,token50,token60).

incomingbond(r00,t110,token30,token40).incomingbond(r00,t110,token50,token60).outcomingb
ond(t110,r00,token70,token80).outcoming(t40,z20,token61).incomingbond(z00,t40,token31,to
ken41).incomingbond(z00,t40,token11,token21).outcomingbond(t40,z20,token11,token21).outc
omingbond(t40,z20,token31,token41).incoming(z10,t40,token61).incoming(z10,t40,token51).o
utcoming(t40,z20,token51)

% GOAL STATE:

placeholds(z20,token51,ts-1).placeholds(z20,token61,ts-
1).placeholdsbond(z20,token31,token41,ts-1).placeholdsbond(z20,token11,token21,ts-1).
```

And the output is:

```
Clingo version 5.4.0

Reading from src/backtracking.Clingo ...

Solving...

UNSATISFIABLE


Models       : 0

Calls        : 1

Time         : 1.031s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)

CPU Time     : 1.030s
```

Command Line Input of Tool for Generating TCRPNs

In this section we present the command line arguments to run the tool.

To run our tool, we write:

```
python3 ./petri_net_producer/main.py --transitions 10 --max_degree 5 > input1
```

where **input1** is the file to redirect the tool's output. We write **–transition 10** to set to 10 the total number of transitions we want the TCRPN to contain. We set the maximum degree to be 5 by writing **--max_degree 5.**

As explained in Chapter 5.5, the number of transitions must be at least double the maximum degree. Otherwise, we get the following message.

```
total transitions must not be less than max degree x 2
```

The output of the program is the same as the input of the simulator we showed in this Chapter.

An example of an input component and bridge can be seen in the file petri_net_modelling_tool/petri_net_producer from the thesis repository found in Appendix.

Bibliography

[1] A. Philippou and K. Psara, “Reversible computation in nets with bonds,” *Journal of Logical and Algebraic Methods in Programming,* vol. 124, 2022.

[2] D. Lindsay, S. Gill, S. D. and e. al., “The evolution of distributed computing systems: from fundamental to new frontiers,” *Computing,* no. 103, pp. 1859-1878, 2021.

[3] C. A. Petri, “Kommunikation mit Automaten,” University of Bonn, 1962.

[4] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE,* vol. 77, no. 4, pp. 541-580, 1989.

[5] M. O'Brien, “Petri Nets: Properties, Applications, and Variations,” University of Pittsburgh.

[6] C. H. Bennet and A. B. Fowler, Rolf W. Landauer (1927-1999) A biographical memoir, vol. 91, washington, d.c.: national academy of sciences, 2009.

[7] M. P. Frank, “Throwing computing into reverse,” *IEEE Spectrum,* vol. 54, no. 9, pp. 32-37, 2017.

[8] V. Lifschitz, Answer Set Programming, Springer Nature Switzerland AG, 2019.

[9] M. Gebser, R. Kaminski, J. Romero and e. al., “Potassco User Guide,” University of Potsdam, 2019.