

Efficient algorithms for temporal logic verification

A look into efficient reachability and tools for temporal logic verification of Petri nets

Master's thesis in Systems, Control and Mechatronics

Olof Olivecrona

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

MASTER'S THESIS: EENX30/2021

Efficient algorithms for temporal logic verification

A look into efficient reachability and tools for temporal logic
verification of Petri nets

OLOF OLIVECRONA



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Efficient algorithms for temporal logic verification
A look into efficient reachability and tools for temporal logic verification of Petri
nets
OLOF OLIVECRONA

© OLOF OLIVECRONA, 2021.

Supervisor & Examiner: Bengt Lennartsson, Department of Electrical Engineering

Master's Thesis 2021
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A transition system representing two synchronised buffers, used in this thesis
as a test example for evaluating reachability algorithms.

Typeset in L^AT_EX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Efficient algorithms for temporal logic verification

A look into efficient reachability and tools for temporal logic verification of Petri nets

OLOF OLIVECRONA

Department of Electrical Engineering

Chalmers University of Technology

Abstract

Many real-world systems can be modelled as discrete event systems (DESs), which have a number of events and discrete states. When an event occurs, the system transits from one state to another. Two ways of representing such systems are transition systems and Petri nets. Finding the reachable states in transition systems is a very central problem within DESs, and solving more complex problems often requires finding them. This makes it very interesting to create an efficient reachability algorithm, which this thesis contributes to by re-implementing a Matlab algorithm in C++ and Python. Interesting differences between the implementations in terms of efficiency are observed. Another important problem within DESs is to verify that systems have certain desirable properties. These properties can be described using temporal logic, where logical formulas can specify not only what should be true in the present, but also what should become true in the future. These specifications can then be verified using nuXmv, which is a symbolic model checker. While transition systems are easy to describe in nuXmv code, Petri nets are not. This thesis presents a parser that translates a Petri net description into nuXmv code, which is shown to greatly reduce the code that the user needs to write. This effectively extends nuXmv such that it may be used for formal verification of Petri nets as well.

Keywords: Discrete event systems, Petri nets, Reachability, Temporal logic verification, nuXmv, Incremental abstraction.

Acknowledgements

I wish to express my gratitude towards Professor Bengt Lennartsson, who single-handedly provided me with the opportunity to carry out this thesis. All work that I have carried out rests entirely on the guidance and assistance he has shown me throughout the process. I would also like to thank my family, as well as my girlfriend, Britt van Zuidam and her family, for their love and support.

Olof Olivecrona, Gothenburg, August 2021

Contents

List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Purpose and motivation	1
1.2 Research questions and contributions	2
1.3 Outline	3
2 Theory	5
2.1 Discrete event systems	5
2.1.1 Transition systems	5
2.1.2 Synchronization of Transition systems	6
2.1.3 Petri nets	8
2.1.4 Synchronisation of Petri nets	9
2.1.5 Translation from Petri net to Transition System	10
2.1.6 Summary	11
2.2 Temporal Logic	11
2.2.1 Linear Temporal Logic	12
2.2.2 Computational Tree Logic	13
2.2.3 Temporal Logic Verification	14
2.2.4 Summary	15
2.3 Incremental abstraction	16
2.3.1 Equivalent states and partitions	16
2.3.2 Invisible transitions and divergence	16
2.3.3 Bisimulation and abstraction	18
2.3.4 Summary	21
2.4 The nuXmv model checker	21
2.4.1 The nuXmv input language	22
2.4.2 Running nuXmv	23
2.4.3 Summary	23
3 Efficient reachability analysis	25
3.1 Representations of Transition Systems	25
3.2 Reachability algorithms	26
3.3 Testing and performance	27
3.3.1 Testing in Matlab	28

3.3.2	Testing in C++	30
3.3.3	Testing in Python	30
3.3.4	Summary	31
4	Translation from Petri net to nuXmv	33
4.1	Features	33
4.2	Syntax	34
4.3	Definition to PetNet code	37
4.4	Performance	38
4.5	Summary	40
5	Comparing nuXmv based model checking with incremental abstraction	41
5.1	Model used for testing	41
5.2	Analytical abstraction	43
5.3	Incremental abstraction	45
5.4	Modelling in PetNet and nuXmv analysis	45
5.5	Results	48
5.6	Summary	48
6	Conclusion	51
6.1	Efficient reachability	51
6.2	PetNet	51
6.3	Incremental abstraction and nuXmv	52
	Bibliography	53

List of Figures

2.1	Two transition systems their synchronous composition.	7
2.2	Two Petri nets and their synchronous composition.	9
2.3	Three very simple Petri nets and their equivalent transition systems.	11
2.4	A sequence of time points that specifies when the propositions p and q are true.	12
2.5	Two finite and two infinite sequences of time points that specifies when the propositions p and q are true.	13
2.6	A tree of time points that specifies when the propositions p and q are true.	14
2.7	Two trees T_1 and T_2 . The proposition p holds for all time points following t_3 in T_2 . Neither p nor q holds in a time point following t_6	14
2.8	The unfair transition system G_{uf}	15
2.9	The fair transition system G_f	15
2.10	A transition system G and two quotient transition systems G_1 and G_2 obtained through the partitions $\Pi_1 = \{\{0, 1\}, \{2, 3\}, \{4, 5, 6\}\}$ and $\Pi_2 = \{\{0, 1\}, \{2\}, \{3\}, \{4\}, \{5, 6\}\}$, respectively.	17
2.11	Several transition systems H_0, \dots, H_8 , used to explain what invisible transitions, stuttering transitions and divergent blocks are.	18
2.12	Incremental abstraction of G_1 , G_2 and G_3	20
2.13	The example transition system G_{main}	22
2.14	The CTL tree equivalent to the transition system G_{main}	22
2.15	Output from nuXmv showing the validity of the two specifications, obtained by running the command <code>nuXmv Gmain.smv</code>	24
3.1	A simple transition system, represented by the matrix in (3.2).	26
3.2	A transition system representing two synchronized buffers. In this figure, M is defined as $N^2 - N$	28
3.3	Visualization of the test results presented in Table 3.1.	29
3.4	Two linear transition systems with states in opposite order.	29
4.1	A bounded Petri net where two processes share a mutual resource r	35
4.2	Petri net example where $i \in [0, 9]$	38
4.3	nuXmv description of the Petri net example from Listing 4.6.	39
5.1	The modular Petri net PN , before synchronisation.	42
5.2	The modular Petri net PN , after synchronisation.	42

5.3	The Petri net NB and F . They are synchronised with PN to create PN_{nb} and PN_f , respectively.	42
5.4	A Petri net consisting of a straight sequence of places as well as the single place Petri net it may be abstracted to if all events but a_0 and a_n are local.	42
5.5	The abstracted Petri net PN^A	44
5.6	The abstracted nonblocking Petri net PN_{nb}^A	44
5.7	The abstracted fair Petri net PN_f^A	44
5.8	The fully abstracted nonblocking Petri net PN_{nb}^{FA}	45
5.9	The fully abstracted fair Petri net PN_f^{FA}	45

List of Tables

2.1	Validity of some example statements for the four sequences shown in Figure 2.5.	13
2.2	CTL formulas and their validity for T_1 and T_2 , shown in Figure 2.7.	14
2.3	Common mathematical symbols and the corresponding symbols in the nuXmv input language.	23
3.1	Execution times in milliseconds from tests in Matlab comparing the three algorithms, as well as a version of source-reach that iterates over states in a random order rather than in ascending order.	28
3.2	Execution times in milliseconds from tests comparing target reach in C++ and Matlab.	30
3.3	Execution times in milliseconds from a test in Python comparing target-reach based on list, target-reach based on numpy.array, as well as BFS-reach.	31
3.4	Execution times in milliseconds from a test of BFS-reach in python.	31
4.1	Valid text for the syntax tags.	35
4.2	Shorthand syntax for actions and definitions of constants and places. Here, p is a place with the limit m and N is a numerical.	36
5.1	Number of characters used to represent various models in PetNet and nuXmv.	48
5.2	Test data for using nuXmv to validate the nonblocking and fairness properties.	48
5.3	Test data for using incremental abstraction to validate the nonblocking and fairness properties.	48

1

Introduction

Many real-world systems can be described as discrete event systems, characterised by having a discrete number of states and events. When an event occurs, the system changes from one state to the next. One straightforward example of such a system could be a light switch, which toggles between its two states (ON and OFF) whenever a certain event occurs, namely whenever its button is pressed. More complicated examples can contain millions of states [1]. As the world has grown increasingly digitised, we find that more and more systems that we wish to model involve discrete quantities. Consider modelling a factory, for example. It would likely be important to know the number of items in buffers, the number of locked doors, and whether industrial robots are busy, idle or in need of repair. These systems are all discrete, and many of the processes inside the factory involve instantaneous changes between states. One such process might be that if all doors are locked and the robot is idle, then it picks an item from a buffer to process it. The number of items in the buffer then instantly decreases by one as the robot changes from being idle to being busy. Other examples include pushing a button, flipping a switch or waiting for the doors of an elevator to open [2].

1.1 Purpose and motivation

The overall goal of this thesis is to contribute to creating more efficient implementations of certain algorithms on discrete event systems. These algorithms do things such as finding all states that a system can reach, verifying that a system has certain desirable properties and combining systems in a process known as synchronisation. Because these systems often get very large, it is interesting to find the most efficient way possible to implement these algorithms. Finding all reachable states is an algorithm of particular importance. For one, it answers verification questions such as "Does the system always manage to avoid this dangerous state?". Furthermore, it is used as a component in other, more complex algorithms such as Sigref, which is used to shrink the number of states in a transition system. This is done in a way that preserves desired properties via a process known as bisimulation minimisation [3]. This thesis will investigate alternative ways to find the reachable states of transition systems with the hopes of creating a more efficient implementation. One promising method, proposed by Bengt Lennartsson, is based on representing the reached states using a boolean vector and finding the one-step-forward reachable states as another such vector. A bitwise OR operation then produces the new reached states. The algorithms discussed so far are based on a model called transition system.

Another model of interest is Petri nets. Petri nets express the same type of systems and can be translated to and from transition systems. Nevertheless, Petri nets have certain advantages over transition systems, such as synchronisation (where two transition systems or Petri nets are joined together) being much more intuitive. NuXmv is a symbolic model checker used for formal verification of transition systems [4]. While it does not support Petri nets, it does support integer variables. By using integers to represent places, it should be possible to model Petri nets. Thus, this thesis intends to produce a parser, converting a simple Petri net description into its equivalent nuXmv code. The parser would enable nuXmv to be used for temporal logic verification on Petri nets, as it is already used for transition systems. Lastly, using the parser together with nuXmv will be tested against another method for temporal logic verification known as incremental abstraction. This other method is based on bisimulation minimisation and the idea of applying it incrementally as the system is synchronised. A detailed description of what this means is given in Chapter 2.3.

1.2 Research questions and contributions

This thesis is concerned with answering the following three questions.

- How can a reachability algorithm be implemented efficiently?
- How can Petri nets be translated into nuXmv code?
- How efficient is nuXmv compared to incremental abstraction as a method for temporal logic verification?

These questions are answered in Chapter 6, and the contribution of this thesis is threefold. Firstly, potential ways to create an efficient way to implement a reachability algorithm are evaluated, and the results are presented. These results show how the choice of language and data structure choice affect performance and how the various algorithms fare against one another. Basing such an algorithm on bitwise boolean operations is shown to be faster in Matlab than in C++ or Python, while an algorithm based on matrix multiplication is shown to be promising for small transition systems. Secondly, this thesis contributes by providing a way to perform temporal logic verification on Petri nets using nuXmv, which it does not currently support in a user-friendly way. This is achieved by developing a parser called PetNet, which translates a simple description of a Petri net into valid nuXmv code. The parser is also shown to greatly reduce the amount of code one must write to model a given Petri net, especially if it contains many places and events. Thirdly, this thesis makes a comparison between two ways of performing temporal logic verification on Petri nets. The first method is based on simplifying the model (while preserving the validity of temporal logic expressions) to the point that the verification step becomes trivial. The second method is based on using PetNet and nuXmv to perform the verification. The former approach is shown to be more efficient. Lastly, a third method, based on simplifying the model by hand before using nuXmv, is shown to reduce the difficulty of the verification significantly.

1.3 Outline

Chapter 2 goes over all theory necessary to understand the thesis. This includes how discrete event systems are modelled as well as what temporal logic is and how it can be used to specify properties that we want our models to have. Two ways of validating such temporal logic specifications are then introduced. The first, known as incremental abstraction, is based on simplifying models while preserving the validity of any specifications. The second is the use of a model checker called nuXmv.

In Chapter 3, several algorithms that find the reachable states of a transition system are described and compared to one another in terms of efficiency.

Chapter 4 goes over the syntax and features of a parser called PetNet. Its function is to translate a simple description of a Petri net into code that the aforementioned model checker nuXmv can understand. Additionally, the PetNet description of a given Petri net is shown to be significantly shorter than its nuXmv code counterpart. Chapter 5 compares the two methods for validating temporal logic specifications that were presented in Chapter 2. A mix between the two is also presented and evaluated, where the model is simplified by hand before being fed into nuXmv. The tests were used to validate the fairness and non-blocking properties of a modular Petri net.

Chapter 6 describes the conclusions drawn from the results that were found and answers the research questions that were raised in Chapter 1.2.

2

Theory

This chapter aims to introduce all theory necessary for understanding the rest of the thesis. Firstly, one must understand what discrete event systems are and how they are modelled. This includes how the models are defined and the important concept of synchronisation, where two models are combined. Next, logic systems for reasoning about time are introduced and used to specify desirable system properties. Lastly, incremental abstraction and nuXmv are presented as methods for verifying that such specifications hold.

2.1 Discrete event systems

Many real-world systems can be described as discrete event systems, characterised by having discrete states and events. When an event occurs, the system changes instantaneously from one state to the next. One straightforward example of such a system could be a light switch, which toggles between its two states (ON and OFF) whenever a particular event occurs, namely whenever its button is pressed. This differs from a dynamic system, where the state changes continuously. Many systems are so-called hybrid systems, which exhibit both continuous and discrete dynamics. For example, the temperature inside a refrigerator has different dynamics depending on whether the door is open or not. The opening and closing of the door could then be modelled as discrete events. Exactly how such models are defined will be this chapter's first concern. Two models, namely transition systems and Petri nets, will now be discussed.

2.1.1 Transition systems

A transition system consists of a set of distinct states that the system can be in and transitions that describe how the system can move from one state to another [1]. These transitions sometimes have so-called event labels such as *openingDoor* and *closingDoor*. The set of all event labels is known as the alphabet. The states can also have labels, but these serve a different purpose. A so-called state label is a set of true propositions in that state, such as $\{doorIsOpen\}$ or $\{doorIsClosed\}$. A transition system must start in some state, namely one of its initial states. There is usually only one possible initial state, but should there be multiple, then the transition system may start in any of them. Having started from some initial state, any state that the transition system can reach by following some sequence of events is known as a reachable state. All other states are said to be unreachable.

A transition system G is defined as a 6 tuple

$$G = \langle X, \Sigma, T, I, AP, \lambda \rangle, \quad (2.1)$$

where

X	is a set of states (also called state space)
Σ	is a finite set of events (also called alphabet)
$T \subseteq X \times \Sigma \times X$	is a transition relation
$I \subseteq X$	is a set of initial states
AP	is a set of atomic propositions
$\lambda : X \rightarrow 2^{AP}$	is a state labeling function

and a transition $(x, a, x') \in T$ is denoted $x \xrightarrow{a} x'$ [5]. For a transition $x \xrightarrow{a} x'$ one refers to x , a and x' as the source state, event label and target state, respectively. Furthermore it is said that G is finite if X is finite. If $\Sigma = \emptyset$, meaning that the transitions lack event labels, then G is sometimes referred to as a Kripke structure.

If AP is defined as $\{m, f\}$, then G may be called an automaton. It is then said that a given state x is marked if $\lambda(x) = \{m\}$, and forbidden if $\lambda(x) = \{f\}$. Marked states are typically desirable, final states that one wants the automaton to be capable of reaching. Forbidden states are the opposite. They are undesirable or dangerous states that the automaton must avoid. Since a marked state should always be reachable, it is interesting to know if that is possible from a given state. It is thus said that a state is coreachable if there exists some sequence of events such that the automaton can reach some marked state from it. Now consider a state that is reachable but not coreachable. Such a state would function as a trap of sorts, as the automaton could reach it and block itself from ever reaching a marked state. It is said that such a state is a blocking state, and it is clear that one would prefer not to have such states [1].

Examples of transition systems are given in Chapter 2.1.2, where they are both defined and visualised.

2.1.2 Synchronization of Transition systems

Synchronising two transition systems is to impose the following rule: If an event a exists in the alphabet of both systems, then that event can only fire in either system if it can fire simultaneously in the other as well. The synchronisation of the two is then a new transition system that captures their combined behaviour. The two original systems are then sometimes referred to as subsystems, while the new, combined system is sometimes said to be modular. Consider two transition systems G_1, G_2 ; both defined as

$$G_i = \langle X_i, \Sigma_i, T_i, I_i, AP_i, \lambda_i \rangle \quad (2.2)$$

for $i = 1, 2$. The synchronization of G_1 and G_2 is then defined as

$$G_1 || G_2 = \langle X_1 \times X_2, \Sigma_1 \cup \Sigma_2, T, I_1 \times I_2, AP_1 \cup AP_2, \lambda \rangle \quad (2.3)$$

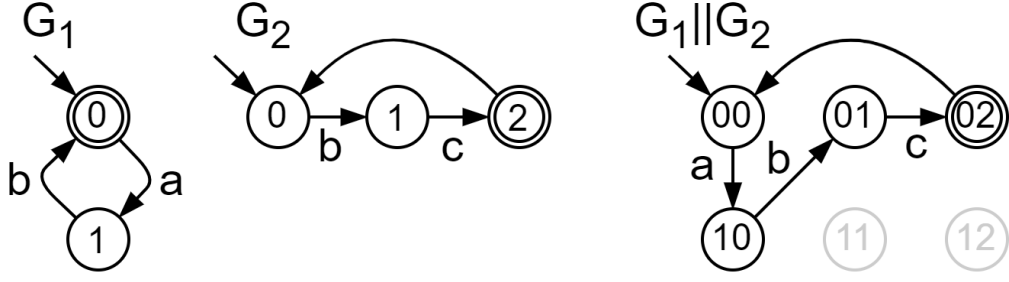


Figure 2.1: Two transition systems their synchronous composition.

where T is defined by

$$\begin{cases} (x_1, x_2) \xrightarrow{a} (x'_1, x'_2) \in T & | \ a \in (\Sigma_1 \cap \Sigma_2), x_1 \xrightarrow{a} x'_1 \in T_1, x_2 \xrightarrow{a} x'_2 \in T_2 \\ (x_1, x_2) \xrightarrow{a} (x'_1, x_2) \in T & | \ a \in (\Sigma_1 \setminus \Sigma_2), x_1 \xrightarrow{a} x'_1 \in T_1 \\ (x_1, x_2) \xrightarrow{a} (x_1, x'_2) \in T & | \ a \in (\Sigma_2 \setminus \Sigma_1), x_2 \xrightarrow{a} x'_2 \in T_2 \end{cases} \quad (2.4)$$

and

$$\lambda : X_1 \times X_2 \rightarrow 2^{AP_1 \cup AP_2} \quad (2.5)$$

[6]. It is worth mentioning that \parallel is a reflexive, commutative and associative operation [1]. Equation 2.4 requires some explanation. The first row states that if the event a is present in both alphabets, and if there exists a pair of transitions $x_1 \xrightarrow{a} x'_1 \in T_1$ and $x_2 \xrightarrow{a} x'_2 \in T_2$, then T has a transition $(x_1, x_2) \xrightarrow{a} (x'_1, x'_2)$. The second and third rows cover the cases where a is only present in Σ_1 and Σ_2 , respectively.

In order to give a better understanding for how transition systems are defined, the two example automata G_1 and G_2 are shown in Figure 2.1 and defined in (2.6) and (2.7), respectively. Their synchronous composition $G_1 \parallel G_2$ is shown in Figure 2.1 and defined in (2.8). Also, the marked states are drawn with double rings, where an alternative would be to write out m next to those states to show that the proposition m holds there. The states $(1, 1)$ and $(1, 2)$ are unreachable and have therefore been drawn in grey to show that they are unimportant and could safely be removed from the model. For a state to be marked in a modular system, all subsystems must be in a marked state. This is why only $(0, 2) \in X_{1 \parallel 2}$ is marked in $G_{1 \parallel 2}$. For all other state labels, it is sufficient that one subsystem is in such a state. This means that if state 0 was forbidden instead of marked in G_1 and G_2 , the states $(0, 0)$, $(0, 1)$, $(0, 2)$ and $(1, 0)$ would all be forbidden in $G_1 \parallel G_2$. Lastly, one should note that the transitions $(2, 0) \in T_2$ and $((0, 2), (0, 0)) \in T$ do not have event labels.

$$\begin{aligned}
 X_1 &= \{0, 1\} & X_2 &= \{0, 1, 2\} \\
 \Sigma_1 &= \{a, b\} & \Sigma_2 &= \{b, c\} \\
 T_1 &= \{(0, a, 1), (1, b, 0)\} & T_2 &= \{(0, b, 1), (1, c, 2), (2, 0)\} \\
 I_1 &= \{0\} & I_2 &= \{0\} \\
 AP_1 &= \{m, f\} & AP_2 &= \{m, f\} \\
 \lambda_1(x) &= \begin{cases} \{m\} & \text{if } x = 0 \\ \emptyset & \text{otherwise} \end{cases} & \lambda_2(x) &= \begin{cases} \{m\} & \text{if } x = 2 \\ \emptyset & \text{otherwise} \end{cases}
 \end{aligned}
 \tag{2.6} \tag{2.7}$$

$$\begin{aligned}
 X_{1||2} &= \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), \} \\
 \Sigma_{1||2} &= \{a, b, c\} \\
 T_{1||2} &= \{((0, 0), a, (1, 0)), ((1, 0), b, (0, 1)), ((0, 1), c, (0, 2)), ((0, 2), (0, 0))\} \\
 I_{1||2} &= \{(0, 0)\} \\
 AP_{1||2} &= \{m, f\} \\
 \lambda_{1||2}(x) &= \begin{cases} \{m\} & \text{if } x = (0, 2) \\ \emptyset & \text{otherwise} \end{cases}
 \end{aligned}
 \tag{2.8}$$

2.1.3 Petri nets

A Petri net consists of a set of places containing tokens, as well as a set of transitions that move the tokens around the net. A place and a transition may be connected by a weighted, directional arc. Two places or two transitions can never be connected. For a transition to fire, the incoming arcs' places must have as many tokens as the weight of that arc. When the event fires, those tokens are removed, and new ones are distributed according to the weights of the outgoing arcs. The decadence and incidence matrices keep track of how many tokens the transitions remove from and add to the places. Just like with transition systems, the transitions may have event labels. The Petri net's state can be said to be the number of tokens currently present in each place. Thus, rather than having an initial state, a Petri net has a so-called initial marking vector that describes each place's initial number of tokens. In fact, by keeping track of the number of tokens in each place as the various transitions fire, one can generate an equivalent transition system. Such a transition system is then known as the Petri net's reachability graph. A Petri net N is defined as a 6 tuple

$$N = \langle P, T, I, D, M, E \rangle \tag{2.9}$$

where

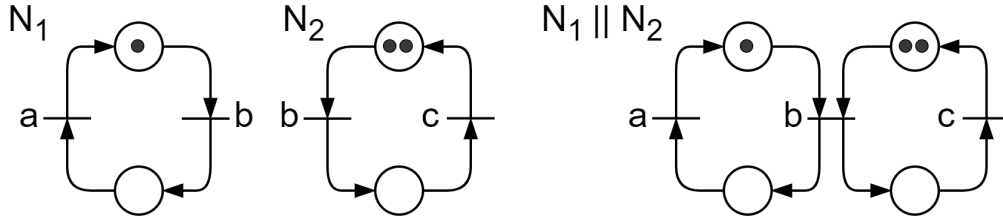


Figure 2.2: Two Petri nets and their synchronous composition.

P is a set of places
 T is a set of transitions
 I is a $n \times m$ incidence matrix
 D is a $n \times m$ decadence matrix
 M is an initial marking vector
 E is a set of event labels

[1]. A bounded Petri net is then defined as

$$N_B = \langle P_B, T_B, I_B, D_B, M_B, E_B, L \rangle \quad (2.10)$$

where $\langle P_B, T_B, I_B, D_B, M_B, E_B \rangle$ is a Petri net, and L is a set of limits that place upper bounds on the number of tokens that each place may have. In such a net, transitions cannot fire if that would cause the number of tokens in a place to exceed its limit. Example Petri nets are defined and visualized in Chapter 2.1.4.

2.1.4 Synchronisation of Petri nets

An advantage of Petri nets is that the synchronisation of two nets is very easy to show visually. Figure 2.2 shows two Petri nets N_1 , N_2 , as well as $N_1 || N_2$. Now consider two arbitrary bounded Petri nets N_1 and N_2 , both defined by

$$N_i = \langle P_i, T_i, I_i, D_i, M_i, E_i, L_i \rangle \quad (2.11)$$

where $i = 1, 2$. The synchronization of N_1 and N_2 is then defined as

$$N_1 || N_2 = \langle P_1 \cup P_2, T, I, D, [M_1^T \ M_2^T]^T, E_1 \cup E_2, L_1 \cup L_2 \rangle \quad (2.12)$$

where T is defined as follows. For each transition with an event only present in either E_1 or E_2 there is a corresponding transition in T with the same label and weighted arcs. For each transition with an event label present in both E_1 and E_2 , there is a corresponding transition in T with the same label and all corresponding weighted arcs from both T_1 and T_2 . The new I and D matrices hold the corresponding weights for the new net [1].

To clarify how Petri nets and their synchronous compositions are defined and visualized, two example Petri nets N_1 and N_2 are presented. These Petri nets are defined in (2.13) and (2.14), respectively, and visualized in Figure 2.2. Their synchronous composition $N_1 || N_2$ is defined in (2.15) and is also visualized in Figure 2.2.

$$\begin{array}{lll}
 P_1 = \{p_1, p_2\} & P_2 = \{p_3, p_4\} & P_{1||2} = \{p_1, p_2, p_3, p_4\} \\
 T_1 = \{t_1, t_2\} & T_2 = \{t_3, t_4\} & T_{1||2} = \{t_1, t_{2,3}, t_4\} \\
 I_1 = \left[\begin{array}{c|cc} & p_1 & p_2 \\ \hline t_1 & 0 & 1 \\ t_2 & 1 & 0 \end{array} \right] & I_2 = \left[\begin{array}{c|cc} & p_3 & p_4 \\ \hline t_3 & 0 & 1 \\ t_4 & 1 & 0 \end{array} \right] & I_{1||2} = \left[\begin{array}{c|cccc} & p_1 & p_2 & p_3 & p_4 \\ \hline t_1 & 1 & 0 & 0 & 0 \\ t_{2,3} & 0 & 1 & 0 & 1 \\ t_4 & 0 & 0 & 1 & 0 \end{array} \right] \\
 D_1 = \left[\begin{array}{c|cc} & p_1 & p_2 \\ \hline t_1 & 1 & 0 \\ t_2 & 0 & 1 \end{array} \right] & D_2 = \left[\begin{array}{c|cc} & p_3 & p_4 \\ \hline t_3 & 1 & 0 \\ t_4 & 0 & 1 \end{array} \right] & D_{1||2} = \left[\begin{array}{c|cccc} & p_1 & p_2 & p_3 & p_4 \\ \hline t_1 & 0 & 1 & 0 & 0 \\ t_{2,3} & 1 & 0 & 1 & 0 \\ t_4 & 0 & 0 & 0 & 1 \end{array} \right] \\
 M_1 = \left[\frac{p_1 \quad p_2}{1 \quad 0} \right] & M_2 = \left[\frac{p_3 \quad p_4}{2 \quad 0} \right] & M_{1||2} = \left[\frac{p_1 \quad p_2 \quad p_3 \quad p_4}{1 \quad 0 \quad 2 \quad 0} \right] \\
 E_1 = \left[\frac{t_1 \quad t_2}{a \quad b} \right] & E_2 = \left[\frac{t_3 \quad t_4}{b \quad c} \right] & E_{1||2} = \left[\frac{t_1 \quad t_{2,3} \quad t_4}{a \quad b \quad c} \right] \\
 L_1 = \left[\frac{p_1 \quad p_2}{1 \quad 1} \right] & L_2 = \left[\frac{p_3 \quad p_4}{2 \quad 2} \right] & L_{1||2} = \left[\frac{p_1 \quad p_2 \quad p_3 \quad p_4}{1 \quad 1 \quad 2 \quad 2} \right] \\
 (2.13) & (2.14) & (2.15)
 \end{array}$$

2.1.5 Translation from Petri net to Transition System

A bounded Petri net can always be translated into a transition system by finding its so-called reachability graph. This is based on the observation that a Petri net's state is defined by the number of tokens in each place. If the Petri net has N places, the current state can be represented by the so-called marking vector m , consisting of N non-negative integers. By systematically firing all possible sequences of transitions and using m to keep track of where the tokens are, a transition system can be generated [1].

An alternative approach is to replace each place with a buffer. Consider a single place with a limit of m , with an event a that adds one token and another event b that removes one. This behaves identically to a buffer transition system with $m + 1$ states, where the event a results in a transition from state n to state $n + 1$ and b results in a transition from state $n + 1$ to state n for $n \in [0, m - 1]$ [7]. The initial number of tokens in the place gives the initial state of the transition system, which would usually be zero. This example is shown in Figure 2.3.

Should the place only have a transition a that adds a token, then the input Petri net N_i can be used instead. Similarly, if a place only has a transition b that removes a token, then the final Petri net N_f may be used. These are also shown in Figure 2.3. In general, any transition that adds tokens could be modelled as increasing the buffer's state, while a transition that removes tokens decreases it instead [7]. However, this method will primarily be used for the three simple cases that have been presented.

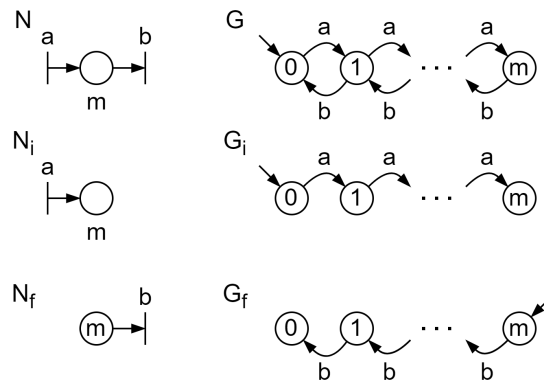


Figure 2.3: Three very simple Petri nets and their equivalent transition systems.

2.1.6 Summary

Transition systems and Petri nets have been introduced as models for discrete event systems. The transition system is based on a set of discrete states and a set of transitions that describe how the system can move between them. The initial state is given. A state may have a label, which is a set of true propositions in that state. An event may also have a label, but such a label is only a symbol and not a set of propositions. Petri nets work differently. A Petri net consists of a set of places that contain tokens and a set of transitions that describe how these tokens can move around the net. A place may have an upper limit on the number of tokens it can hold, and if all places have such limits, then the net is said to be bounded. Transitions may have labels, just like in transition systems. Both models describe the same types of behaviour, and it is possible to translate from one to another. Translating Petri nets to transition systems has been described as that is of importance to this thesis. The concept of synchronisation has also been described for both models, where two systems are combined. This is done by restricting shared events to only fire if they can fire in both systems simultaneously.

2.2 Temporal Logic

A critical tool when working with any system is the ability to verify that it works as it should. In the context of discrete event systems, this means guaranteeing that certain things will or will not become true in the future. A traffic light should always eventually turn green, for example, and two industrial robots should never enter the same space simultaneously (because they would then crash into one another). Temporal logic allows us to express such properties. There are several versions of temporal logic, but only the most common are of interest, namely linear temporal logic (LTL) and computational tree logic (CTL). The former shall be considered first.

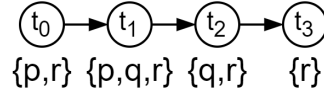


Figure 2.4: A sequence of time points that specifies when the propositions p and q are true.

2.2.1 Linear Temporal Logic

In linear temporal logic (LTL), time is represented by a set $T = t_0, t_1, t_2, \dots$ and a total order $<$. The comparison $t_a < t_b$ then means that t_a occurs before t_b . For simplicity it is assumed that t_0 is the first time point, t_1 the second and so on. This gives us an ordered sequence of time points where $t_0 < t_1 < t_2 < \dots$ [8]. This discrete representation of time must now be able to specify when things are true. This is simple. For a set AP of atomic propositions, it is specified at which time points those propositions are true. Say for example that one has $T = \{t_0, t_1, t_2, t_3\}$, $AP = \{p, q, r\}$. It can then be specified that p is true in t_0, t_1 , that q is true in t_1, t_2 and that r is true at every time point in T . This example is visualised in Figure 2.4. If one imagines starting in the initial time point t_0 , one could say things such as

- " p is true in the next time point"
- "At some point in the future, q is true"
- " r is always true"
- " p is true until q becomes true"

and indeed, the ability to make these kinds of statements was the whole point. Operators for them are thus introduced as such [9].

- $\bigcirc p$ "next p "
- $\Diamond q$ "eventually q "
- $\Box r$ "always r "
- pUq " p until q "

To further clarify what these operators mean, Figure 2.5 shows four sequences of time points while Table 2.1 shows the validity of the most common LTL formulas. The sequences T_3 and T_4 are both infinite so some explanation of when p holds is needed. p holds in every odd-numbered time point in T_3 , and in every time point occurring after t_2 in T_4 .

The formal syntax of an LTL formula is now defined as

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 U \varphi_2 \quad (2.16)$$

For those unfamiliar with the notation, this definition should be interpreted as: "An LTL formula φ is defined as an atomic proposition p , or as the negation of some formula, or as the conjunction between two formulas, or as the next operator applied to some formula or as the until operator applied to two formulas". One might notice that some statements are missing from this definition, such as the disjunction between two formulas. However, these can be defined in terms of what has already been defined. For example, disjunction, eventually and always may respectively be

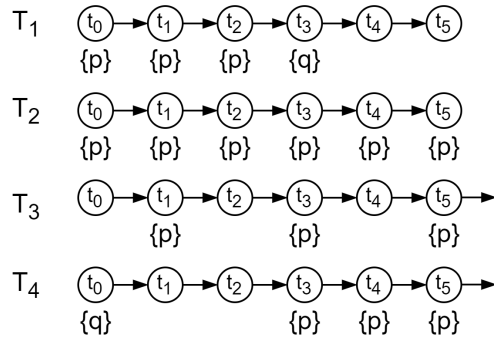


Figure 2.5: Two finite and two infinite sequences of time points that specifies when the propositions p and q are true.

	T_1	T_2	T_3	T_4
p	true	true	false	false
$\bigcirc p$	true	true	true	false
$\Diamond p$	true	true	true	true
$\Diamond q$	true	false	false	true
$\Box p$	false	true	false	false
pUq	true	false	false	true
$\Box \Diamond p$	false	true	true	true
$\Diamond \Box p$	false	true	false	true

Table 2.1: Validity of some example statements for the four sequences shown in Figure 2.5.

defined as

$$\varphi_1 \vee \varphi_2 := \neg(\neg\varphi_1 \wedge \neg\varphi_2) \quad (2.17)$$

$$\Diamond\varphi := \top U \varphi \quad (2.18)$$

$$\Box\varphi := \neg\Diamond\neg\varphi \quad (2.19)$$

As Figure 2.4 hints at, the idea is to apply LTL to transition systems, where time moves forward as the system transitions from one state to the next. A statement such as "a marked state is always reachable" could then be rephrased as "It is always true that, eventually m will be true" and written in LTL as $\Box\Diamond m$. However, transition systems usually have many possible sequences of events, motivating the need for specifications such as "There exists a sequence of events where p is always true". To handle these types of statements, one must stop thinking of time as linear and start thinking about it as a tree.

2.2.2 Computational Tree Logic

In computational tree logic (CTL), timepoints are not ordered as a straight sequence but as a tree. While it can be strange to think of time as anything but linear, one can imagine the tree describing many possible linear paths that the future could take. A time branch is one such possible path, and one could write an LTL formula to specify something along it. Consider the tree shown in Figure 2.6. There exists a time branch (namely $\{t_0, t_1, t_4\}$) where the LTL formula $\Box p$ holds. Also, for every time branch, the LTL formula $\Diamond q$ holds. Essentially, the following two new statements must be expressible.

1. "In all time branches, this LTL formula holds"
2. "There exists a time branch where this LTL formula holds"

For this purpose, two additional operators are introduced [9].

$\forall\varphi$ for each time branch, φ holds

$\exists\varphi$ there exists a time branch such that φ holds

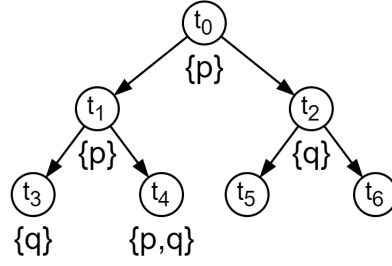


Figure 2.6: A tree of time points that specifies when the propositions p and q are true.

φ	T_1	T_2
$\exists \Diamond p$	true	true
$\exists \Diamond q$	true	true
$\exists \Box p$	true	true
$\exists \Box q$	false	false
$\forall \Diamond p$	true	false
$\forall \Diamond q$	true	false
$\forall \Box p$	true	false
$\forall \Box q$	false	false

Table 2.2: CTL formulas and their validity for T_1 and T_2 , shown in Figure 2.7.

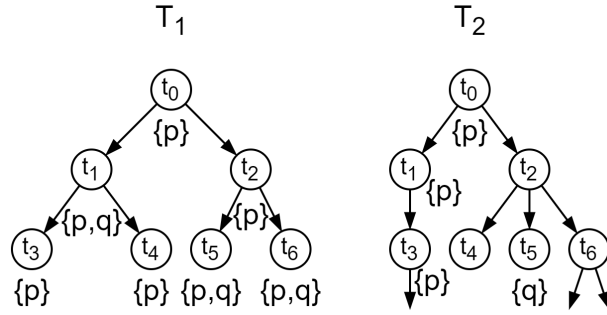


Figure 2.7: Two trees T_1 and T_2 . The proposition p holds for all time points following t_3 in T_2 . Neither p nor q holds in a time point following t_6 .

As before, the formal syntax a CTL formula can be defined as follows:

$$\varphi ::= p \mid \neg \varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists \Box \varphi \mid \exists (\varphi_1 U \varphi_2) \mid \exists \bigcirc \varphi \quad (2.20)$$

Table 2.2 shows the validity of the most common CTL formulas for the two trees displayed in Figure 2.7. One may notice that T_2 contains some infinite sequences. In the left sequence, p holds for all time points occurring after t_3 . In the right sequences, neither p nor q are true at any time point after t_6 . Now that the tools to express when statements are true have been acquired, the next goal is to formulate LTL and CTL specifications and understand how those may be verified.

2.2.3 Temporal Logic Verification

Once the desirable behaviour of a transition system has been specified, one would like to verify that the specification holds. More specifically, the specification must hold in the initial state(s). This can be done using μ calculus [10], but exactly how that works is beyond the scope of this thesis. Instead, the focus lies on understanding how to write specifications in temporal logic. Later, two verification methods will be introduced, namely incremental abstraction and the nuXmv model checker.

The two most important properties that are to be verified are the non-blocking property and the fairness property. Non-blocking means that the transition system

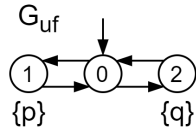


Figure 2.8: The unfair transition system G_{uf} .

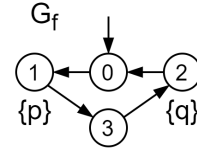


Figure 2.9: The fair transition system G_f .

is free from blocking states and can be written as the following CTL formula

$$\varphi_{nb} = \forall \Box \exists \Diamond m, \quad (2.21)$$

where the atomic proposition m holds in the marked states.

The concept of fairness requires a little more explanation. Consider the simple transition system G_{uf} shown in Figure 2.8. Consider the possible sequence of states $0, 1, 0, 1, \dots$, where p is always eventually true but q never becomes true. This is unfair in the sense that state 1 steals all the attention while state 2 starves. The fairness condition can be written formally as the following LTL formula

$$\varphi_f = \Box \Diamond p \wedge \Box \Diamond q. \quad (2.22)$$

The problem may be resolved by adding another state to obtain the fair transition system G_f shown in Figure 2.9. Here the only possible sequence of states is $0, 1, 3, 2, 0, \dots$ where p and q are both always eventually true [9].

It is important to note that proof of fairness is an infinite sequence of states. In the example above, the proof was the infinite loop $0, 1, 3, 2, 0, \dots$ where p and q become true. Sometimes, a finite sequence is sufficient. If one wants to prove that $\exists \Diamond p$ holds, for example, the sequence $0, 1$ is sufficient. An expression such as $\forall \Diamond p$ is proven differently. Instead of proving it directly, one would try to find a counterexample that disproves it. If no counterexample exists, then the statement must be true. For large transition systems, finding these proofs can be very computationally intensive. This problem is dealt with in the next section, where it is shown how one can avoid verifying properties on large systems. The method is based on the observation that two different transition systems can have the same validity of temporal logic formulas. The goal is then to create a simpler system, equivalent to the large one in the sense that temporal logic formulas hold on the large system if and only if they also hold on the simple one.

2.2.4 Summary

The ability to verify that systems have certain desirable properties (such as non-blocking and fairness) is essential. These properties are specified using two types of temporal logic, namely linear temporal logic (LTL) and computational tree logic (CTL). In LTL, time is represented by a sequence of time points, ordered from first to last. Instead of propositions simply being true or false, they may now be true at only some time points. One can then write formulas that specify what should be true at the first time point and what should become true in the future. The

next, eventually, always and until operators are used for this. CTL represents time as a tree instead, which can be imagined as many possible (linear time) sequences that could happen. The for all and there exists operators are introduced to specify whether something should hold in all or at least one of these possible futures. It has also been shown that proof of a temporal logic specification is given as a finite or infinite sequence of states or the lack thereof.

2.3 Incremental abstraction

Verifying temporal logic specifications can take a lot of time for large transition systems. Unfortunately, many transition systems are enormous, usually consisting of many smaller transition systems (called subsystems) that have been synchronised. This issue shall be dealt with by abstracting (simplifying) the subsystems before and after they are synchronised without affecting the validity of important temporal logic specifications in a process called incremental abstraction. The final, synchronised system will then be much simpler, and the specifications may be validated on it instead of on the original system, saving a lot of computation time.

2.3.1 Equivalent states and partitions

The idea is to simplify transition systems by grouping states that are considered equivalent. Thus, notation for describing that two states are equivalent will now be introduced. Consider a transition system $G = \langle X, \Sigma, T, I, AP, \lambda \rangle$. If two states $x, y \in X$ are considered equivalent, then that is denoted as $x \sim y$. By dividing X into groups of equivalent states, a partition Π of X is created. Formally, Π is a set of nonempty subsets of X where each element $x \in X$ is in exactly one of these subsets. These subsets (or groups) are then known as block states, and $\Pi(x)$ refers to the block state that $x \in X$ is part of. Given two partitions Π_1, Π_2 it is said that Π_1 is finer than Π_2 and that Π_2 is coarser than Π_1 if $\Pi_1(x) \subseteq \Pi_2(x)$ for all $x \in X$ and this is denoted as $\Pi_1 \preceq \Pi_2$. The finest possible partition would then be obtained by grouping the states individually, while the coarsest possible would group every state into one. Thus, a coarser partition translates to a greater simplification [9].

Given a transition system G and a partition Π , a simplified transition system $G_{/\sim} = \langle \Pi, \Sigma, T_{\Pi}, I_{\Pi}, AP, \lambda_{\Pi} \rangle$ is known as a quotient transition system. Such a system may be created in the following way. The states X are replaced with the partition Π , while the alphabet Σ remains the same. For each transition $x \xrightarrow{a} y \in T$ in G there is then a corresponding block transition $\Pi(x) \xrightarrow{a} \Pi(y) \in T_{\Pi}$ in G_{Π} . The blocks that contain initial states become the initial block states. The atomic propositions remain the same. The block labeling function $\lambda_{\Pi} = \lambda(x)$ assumes that the states in each block share label, i.e. that $\Pi(x) = \Pi(y) \implies \lambda(x) = \lambda(y)$ [9].

2.3.2 Invisible transitions and divergence

Consider the example transition system shown in Figure 2.10, as well as the partitions $\Pi_1 = \{\{0, 1\}, \{2, 3\}, \{4, 5, 6\}\}$, $\Pi_2 = \{\{0, 1\}, \{2\}, \{3\}, \{4\}, \{5, 6\}\}$. This allows for the creation of two quotient transition systems, also shown in Figure 2.10. An

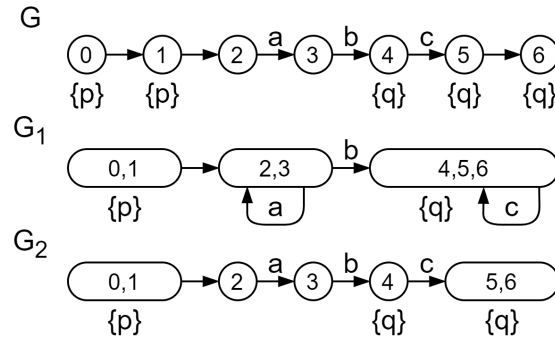


Figure 2.10: A transition system G and two quotient transition systems G_1 and G_2 obtained through the partitions $\Pi_1 = \{\{0, 1\}, \{2, 3\}, \{4, 5, 6\}\}$ and $\Pi_2 = \{\{0, 1\}, \{2\}, \{3\}, \{4\}, \{5, 6\}\}$, respectively.

issue with G_1 is that some temporal logic specifications that hold for G (such as $\Diamond q$) do not hold for G_1 . On the other hand, G_2 preserves all *CTL* expressions (and thus implicitly also all *LTL* expressions) except for those including the next operator \bigcirc . This operator cannot be accommodated, but fortunately, that does not reduce expressiveness.

The overall goal of simplifying transition systems can now be stated more precisely. The idea is to create a partition Π that is as coarse as possible while not affecting the validity of temporal logic specifications. However, only doing that would not be enough. Consider, for example, a system consisting of the synchronisation of two subsystems. The plan is to first abstract, then synchronise. It is then vital that the abstraction does not remove events that are present in other subsystems. Only events that are not shared with other subsystems, called local events, may be removed. These events are hidden by replacing the event labels of local events with the special τ label. This τ label will then help us to know which transitions that can be abstracted [5].

For a given transition system G and partition Π , consider a transition in G with a τ label that does not move the transition system from one block to another. Such a transition is said to be invisible. A transition that is not invisible is said to be visible. If a series of invisible transitions end in a single visible one $x_0 \xrightarrow{\tau} x_1 \xrightarrow{\tau} \dots \xrightarrow{\tau} x_n \xrightarrow{a} x'$, then it is said to be a stuttering transition, denoted as $x \rightarrow x'$.

An issue known as divergence will now be introduced. Consider a block state that contains a loop of invisible transitions. The transition system could then effectively halt in that state, stuck in an infinite loop. Even though these transitions are invisible, they should not be abstracted since doing so would change the model's behaviour. To protect the states involved in the loop from being removed, a special atomic proposition \hookrightarrow is added to mark them divergent. Since all states inside a block should have the same label, this creates a divergent block. The invisible transitions can now be removed. Finally, a simple self-loop $B \xrightarrow{\tau} B$ is added to any block B which has the \hookrightarrow label. This makes the abstraction divergence sensitive since it preserves divergent behaviour [11].

Consider the transition systems H_0, \dots, H_8 shown in Figure 2.11. Consider further wanting to synchronise H_0 with H_1 . One can then notice that since a is only present

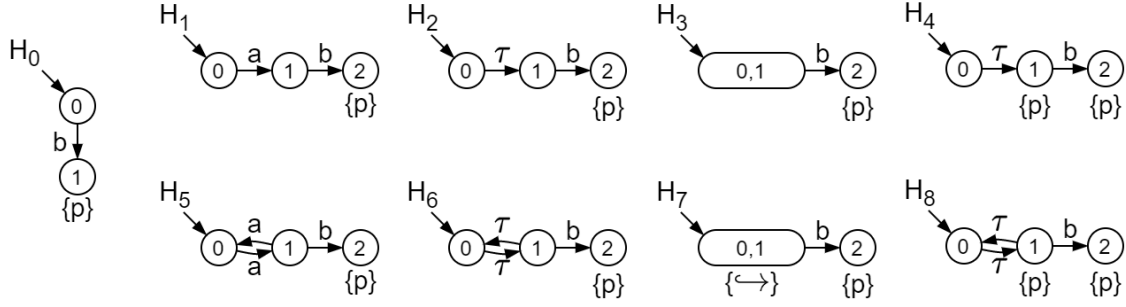


Figure 2.11: Several transition systems H_0, \dots, H_8 , used to explain what invisible transitions, stuttering transitions and divergent blocks are.

in H_1 , it is a local event and may be replaced with τ , thus turning the transition system into H_2 . Now there is a local event between two states with the same label (no label at all). This means that τ is an invisible transition and we can group state 0 and 1 together to form H_3 . Note that H_4 contains no invisible transitions and cannot be made simpler. Consider now wanting to synchronise H_0 with H_5 . Again, a is local, giving us H_6 . Both $(0, \tau, 1)$ and $(1, \tau, 0)$ are invisible, but since they form a loop the resulting block state is divergent, as shown in H_7 . Similarly to H_4 , H_8 cannot be made simpler as it contains no invisible transitions. Lastly, one can note that H_2 has a stuttering transition $0 \xrightarrow{b} 2$.

When two subsystems are synchronised, shared events are synchronised while local events are not. However, since the event labels of local events have been replaced with τ , they will be treated as if they were shared. This is not intended. Clearly, any events with the τ label should be treated as local during the synchronisation. Thus synchronisation is now redefined to take this into account. For two transition systems

$$G_i = \langle X_i, \Sigma_i, T_i, I_i, AP_i, \lambda_i \rangle \quad (2.23)$$

where $i = 1, 2$, the synchronization of G_1 and G_2 is now redefined as

$$G_1 || G_2 = \langle X_1 \times X_2, \Sigma_1 \cup \Sigma_2, T, I_1 \times I_2, AP_1 \cup AP_2, \lambda \rangle \quad (2.24)$$

where T is defined by

$$\begin{cases} (x_1, x_2) \xrightarrow{a} (x'_1, x'_2) \in T \mid a \in (\Sigma_1 \cap \Sigma_2) \setminus \{\tau\}, x_1 \xrightarrow{a} x'_1 \in T_1, x_2 \xrightarrow{a} x'_2 \in T_2 \\ (x_1, x_2) \xrightarrow{a} (x'_1, x_2) \in T \mid a \in (\Sigma_1 \setminus \Sigma_2) \cup \{\tau\}, x_1 \xrightarrow{a} x'_1 \in T_1 \\ (x_1, x_2) \xrightarrow{a} (x_1, x'_2) \in T \mid a \in (\Sigma_2 \setminus \Sigma_1) \cup \{\tau\}, x_2 \xrightarrow{a} x'_2 \in T_2 \end{cases} \quad (2.25)$$

2.3.3 Bisimulation and abstraction

The term bisimulation implies the existence of the term simulation. This term indeed exists, but since the focus lies in bisimulations, only a conceptual explanation will be given.

Consider wanting to flip a coin but only having a six-sided die available. Flipping a coin would cause it to enter one of two possible states, say $S_C = \{s_{heads}, s_{tails}\}$.

The dice on the other hand, enters one of six possible states, $S_D = \{s_1, s_2, \dots, s_6\}$. In order to use the dice as if it were a coin, one could decide that a roll of 1-3 counts as heads, and 4-6 counts as tails. In more formal terms, one could create a partition $\Pi = \{\{s_1, s_2, s_3\}, \{s_4, s_5, s_6\}\}$ and an equivalence relation \sim between the blocks states in Π and the states in S_D such that $s_{heads} \sim s_i$ for $i = 1, 2, 3$ and $s_{heads} \sim s_j$ for $j = 4, 5, 6$. The key here is that while a die is certainly not the same as a coin, it can simulate a coin.

Obviously, the partition Π and relation \sim cannot be used to allow a coin to simulate a die. The simulation is one-directional. If the simulation is bidirectional, meaning that both simulate each other, then it is a bisimulation instead. At last, a divergent sensitive visible bisimulation can be defined.

Given two transition systems

$$G_i = \langle X_i, \Sigma_i, T_i, I_i, AP_i, \lambda_i \rangle \quad (2.26)$$

where $i = 1, 2$, and a set of block transitions

$$T_\Pi(x) = \{B \xrightarrow{a} B' \mid B, B' \in \Pi \wedge x \in B \wedge (\exists x' \in B') \mid x \xrightarrow{a} x'\} \quad (2.27)$$

a partition Π that satisfies

$$\Pi(x) = \{y \in X \mid \lambda(x) = \lambda(y) \wedge T_\Pi(x) = T_\Pi(y)\} \quad (2.28)$$

is a visible bisimulation equivalence and the states $x, y \in \Pi(x)$ are said to be visibly bisimilar, denoted $x \sim_v y$. If it is assumed that the special \hookrightarrow label has been used to mark divergent states, then the equivalence is divergent-sensitive, giving us divergent-sensitive visible bisimulation equivalence (DSVB) [5].

Finally, what it means to simplify a transition system shall now be defined. Consider a transition system G and a partition Π . If $x \sim \Pi(x)$ for all $x \in X$, then $G \sim G_{/\sim}$. By hiding the local events in G and generating the quotient transition system, an abstracted transition system is obtained. If the transition system consists of several subsystems, they should be abstracted before they are synchronised. After synchronising two abstracted subsystems, the resulting system is also abstracted. As some systems are synchronised, more transitions become local and further abstractions are made. The process of abstracting and synchronising then continues until all subsystems have been included.

Figure 2.12 shows incremental abstraction applied to the three subsystems G_1 , G_2 and G_3 . The alphabets for the three systems are $\Sigma_1 = \{a_1, a_2\}$, $\Sigma_2 = \{a_2, a_3\}$ and $\Sigma_3 = \{a_3, a_4\}$. Thus, a_1 is local to G_1 and is abstracted in step A1. G_2 has no local events so step A2 has no effect. When these two systems are synchronised in step S1, the event a_2 becomes local to $H_1 \parallel H_2$ and is subsequently abstracted in step A3. Step A4 abstracts a_4 , local to G_3 . After the final synchronisation, all events become local and a final abstraction results in the transition system L .

One might point out that it would be simpler to first synchronise all subsystems together and then abstract the result only once. Indeed, this would be simpler and also generate the exact same final result. The issue with this is that the synchronised system could be enormous, even for a moderate number of relatively small subsystems. By abstracting incrementally, the model is kept small, which allows work to continue much faster.

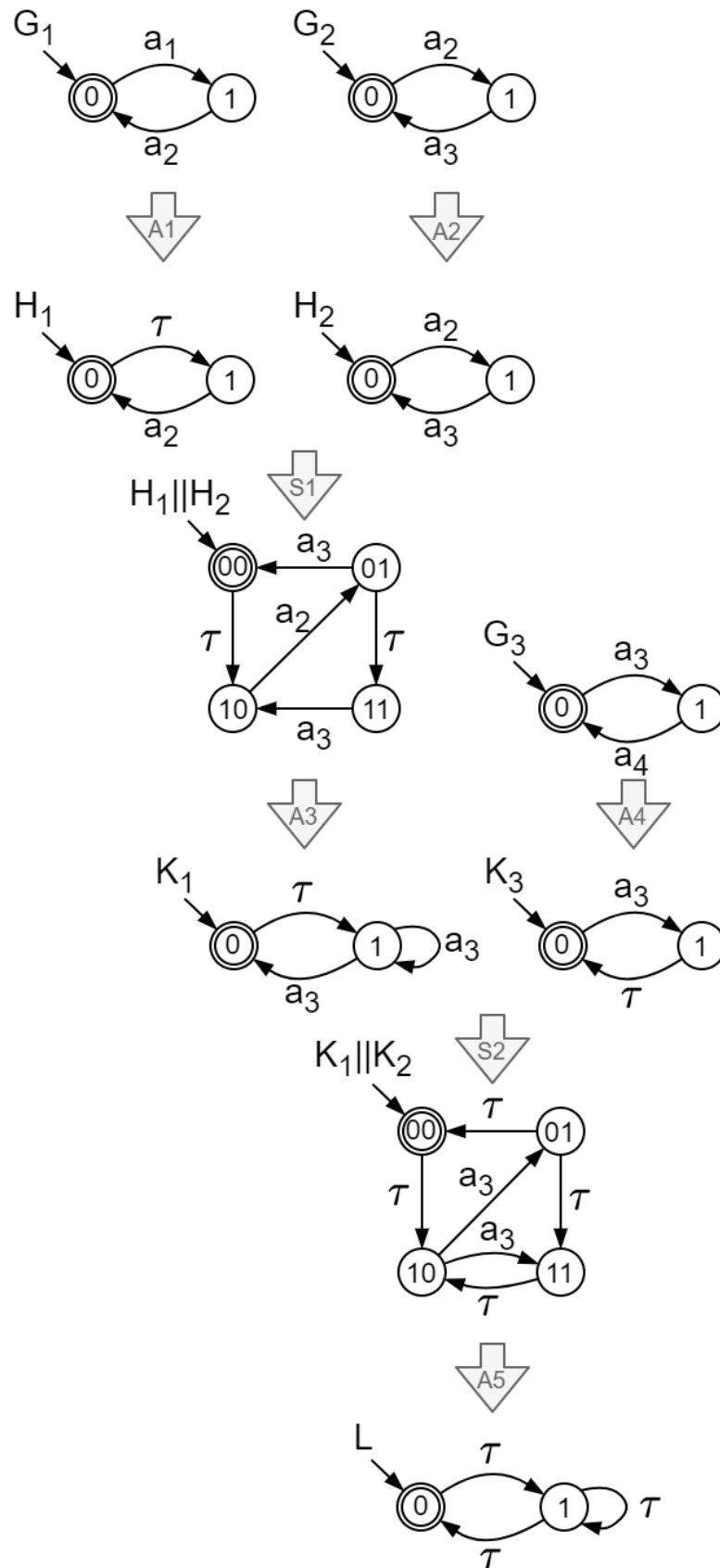


Figure 2.12: Incremental abstraction of G_1 , G_2 and G_3 .

The whole point of this was to simplify in such a way that temporal logic is preserved, which it does if the equivalence used when abstracting is a DSVB equivalence \sim_d . Then, any temporal logic specification holds on G/\sim_d if and only if it also holds on G . This means that specifications may be verified on G/\sim_d instead of on G , which is much easier. Often, G/\sim_d only contains a few states, making the verification step trivial. This is seen in Figure 2.12, where the abstracted model L only contains two states and is clearly non-blocking. Importantly, this implies that $G_1||G_2||G_3$ is also non-blocking.

2.3.4 Summary

The idea of equivalent states has been introduced and used to generate quotient transition systems, where equivalent states are grouped into blocks. An equivalence that generates the smallest possible quotient transition system that still preserves the validity of temporal logic specifications was sought. Local transitions were introduced, characterised by not being shared by other subsystems. Furthermore, local transitions that do not move the transition system from one block to another have been introduced as invisible transitions. These are often safe to remove, but if a block contains a loop of such transitions, then that block is divergent. In theory, the system could halt in such a block as it loops indefinitely and never leaves. It has been shown how a system could simulate another, and this concept has been used to define a divergent sensitive visible bisimulation equivalence (DSVB equivalence), which preserves temporal logic. Abstraction has been defined and used to form incremental abstraction, where subsystems are abstracted before and after they are synchronised. This reduction then results in a quotient transition system that can be used to verify temporal logic specifications very easily.

2.4 The nuXmv model checker

Given a transition system G and a temporal logic specification φ , proving that φ holds on G is a tedious process best left to computers. As shown in Chapter 2.2.3, such a proof consists of a sequence of states. Sometimes, a finite sequence is sufficient, such as proving that $\exists\Diamond p$ holds by showing a sequence ending in a state where p is true. Other times, the sequence needs to be infinitely long. One example would be a proof of $\Box p$ by showing an infinite loop of states where p is true. In any case, the model checker nuXmv can perform the verification for us. This section is intended to explain what nuXmv is, how it is used, and highlight why the software tool is not suitable for Petri nets. One should note that nuXmv is an extension of the nuSMV model checker and that most of what is discussed in this chapter also applies to nuSMV [12]. However, since only nuXmv was used, only nuXmv will be referred to. One should also keep in mind that nuXmv has many more features than those described in this chapter. The information presented here is only intended to give a sufficient understanding for reading the rest of the thesis.

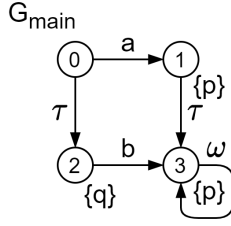


Figure 2.13: The example transition system G_{main} .

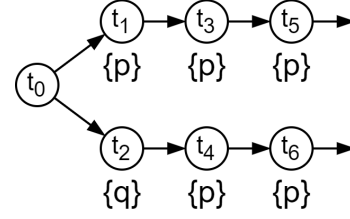


Figure 2.14: The CTL tree equivalent to the transition system G_{main} .

2.4.1 The nuXmv input language

NuXmv is a model checker used to verify LTL and CTL specifications on transition systems with a finite number of states and transitions [13]. The transition systems and specifications must be written in the nuXmv input language, which will now be described. Consider the transition system G_{main} shown in Figure 2.13. The transition system consists of two possible paths, one where p becomes true and another where q first becomes true. Both paths end in the final state 4, which has a self-loop with an ω label. The purpose of this transition is to model that 4 is a terminal state while still letting G_{main} be free from deadlocks, which can cause issues for nuXmv. Consider now that one wishes to verify the specification $\varphi_1 = \Diamond p$ as well as $\varphi_2 = \exists \Diamond q$. The equivalent CTL tree of G_{main} is shown in Figure 2.14. One can notice that φ_2 is a CTL expression and that it holds. However, φ_1 is an LTL expression even though time is not linear here. NuXmv interprets this as the CTL expression $\forall \varphi_1$, which is also true since both sequences eventually reach a p state. G , φ_1 and φ_2 can be described with the following nuXmv input code.

```

MODULE main
VAR
  x : 0..3;
DEFINE
  p := x=1 | x=3;
  q := x=2;
IVAR
  event : {a, b, tau, omega};
INIT
  x=0;
TRANS
  (event=a & x=0 & next(x)=1) |
  (event=tau & x=0 & next(x)=2) |
  (event=tau & x=1 & next(x)=3) |
  (event=b & x=2 & next(x)=3) |
  (event=omega & x=3 & next(x)=3);
LTLSPEC F p
SPEC EF q
INVARSPEC !(p & q)

```

Listing 2.1: PetNet code for G_{main} shown in Figure 2.13.

The description of the transition system begins with the **MODULE** keyword, followed by the transition systems name. To describe a transition system consisting of multiple

Mathematical symbol	\neg	\wedge	\vee	\bigcirc	\diamond	\square	U	\forall	\exists
nuXmv symbol	!	&		X	F	G	U	A	E

Table 2.3: Common mathematical symbols and the corresponding symbols in the nuXmv input language.

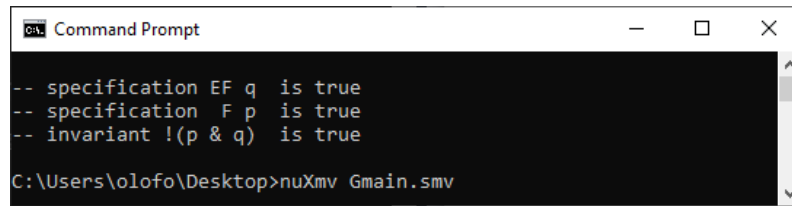
subsystems, one could write several modules. Under the **VAR** (variables) keyword, the state space is defined. In this example, the integer variable x can assume the values 0, 1, 2 and 3. The **DEFINE** keyword allows for the definition of symbols in order to make the code easier to read. In this example, it is used to implement the labelling function. Since p should be true when $x=2$, for example, it is defined as such. If p would be true for multiple states, say 2 and 4, one could write that as $p := x=2 \mid x=4$. Under the **IVAR** (input variables) keyword, the alphabet is defined. The initial state is defined under the **INIT** (initial) keyword as a boolean expression that is true initially. Should there be more than one initial state, one would need some boolean expression that is true in all initial states. If, for example 2 was also an initial state, one could write $x=1 \mid x=2$. The transitions are defined under the **TRANS** (transitions) statements as a boolean expression, which will be written to follow an and-or structure. In order to make a transition, nuXmv will try to satisfy the expression. Because of the structure, it is then sufficient to satisfy any single row. If $x=2$ for example, then only row three could be satisfied, which would require the event **tau** as well as letting the next value of x be 4. Note that the next value of all variables must be specified, or they will be considered unconstrained and be able to assume any value inside their domain. This fact will be important to remember in Chapter 4, as it presents a problem for the modelling of Petri nets. Specifications are typically written using either LTL or CTL, under the **LTLSPEC** (LTL specification) and **SPEC** (CTL specification) keywords respectively. A third alternative is to write invariant specifications under the **INVARSPEC** keyword. Invariants are simply propositions that must always hold. The most common logical symbols and the syntax for them in the nuXmv input language are shown in Table 2.3 [14].

2.4.2 Running nuXmv

The code may be written as a simple text and saved as a .smv file, for example, **Gmain.smv**. Running the code is then very simple. Once nuXmv has been installed, it can be run from the command line by navigating to the folder where the file is stored and using the simple command **nuXmv Gmain.smv**. nuXmv will then print whether the specifications are true or false and provide counterexamples for false specifications [4]. Running the G_{main} example produces the output shown in Figure 2.15. All specifications are found to be true, so no counterexamples exits.

2.4.3 Summary

The nuXmv model checker is used for the verification of temporal logic specifications on transition systems. To use it, one must describe the transition system and the



```
Command Prompt
-- specification EF q is true
-- specification F p is true
-- invariant !(p & q) is true
C:\Users\olofo\Desktop>nuXmv Gmain.smv
```

Figure 2.15: Output from nuXmv showing the validity of the two specifications, obtained by running the command `nuXmv Gmain.smv`.

specifications in the nuXmv input language. The state space and alphabet are declared first as variables, followed by a boolean statement that holds in the initial condition. The transitions are then described using a boolean expression that, for each event, describes both what the current state is and what the next state should be. If an event does not specify what the next value should be for a given variable, then it may assume any value within its domain when that event occurs. NuXmv can be run from the command line and will then try to find counterexamples that disprove the specifications. If counterexamples are found, then they will be printed. If a specification holds, no such proof is given.

3

Efficient reachability analysis

Finding the reachable states of a transition system is a central problem within discrete event systems for two main reasons. Firstly, it is often interesting to know which states are reachable. Secondly, solving more complex problems often requires finding the reachable states, sometimes several times over [3]. Because of this, it is very interesting to evaluate ways to implement a reachability algorithm that is as efficient as possible. One promising implementation, proposed by Bengt Lennartsson, exploits Matlabs efficiency on performing vectorial operations. This implementation shall be used as a starting point. Then, we shall compare the algorithm against modifications of itself and implementations in C++ and Python. However, we must first describe how we choose to represent transition systems.

3.1 Representations of Transition Systems

Consider a transition system $G = \langle X, \Sigma, T, I, AP, \lambda \rangle$. Since the goal is just to find the reachable states, the alphabet Σ , the propositions AP and the labeling function λ can be omitted. Only the states X , the initial states I as well as the transitions T must be taken into account. The transitions can be considered unlabelled. For simplicity, it is assumed that the states are ordered from 1 to N , where N is the number of states. The state space is thus $X = \{1, 2, \dots, N\}$. The initial states can then be represented by a binary vector $x_I = [x_{I,1} \ x_{I,2} \ \dots \ x_{I,N}]$ where

$$x_{I,k} = \begin{cases} 1 & \text{if } x = k \text{ is an initial state} \\ 0 & \text{otherwise} \end{cases} \quad (3.1)$$

Any reachability algorithm necessarily has to keep track of the states that have been reached. The reached states will be represented in the same way as the initial states, namely using a binary vector x of length N . N is then the number of states and if $x_i = 1$ that means that x_i has been reached. Furthermore, $x = x_I$ initially, thus marking the initial states as having been reached from the start. All algorithms in this thesis use these representations of the states, the initial states and the reached states. However, they differ in their representations of the transitions, which will be described now.

Consider the simple transition system G shown in Figure 3.1. A possible representation of the transitions is shown in (3.2) as a 2×4 matrix.

$$\delta = \begin{bmatrix} 2 & 4 & 4 & 0 \\ 3 & 0 & 0 & 0 \end{bmatrix} \quad (3.2)$$

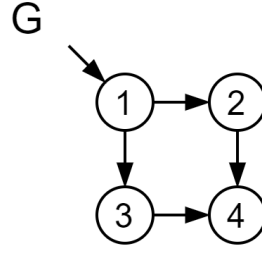


Figure 3.1: A simple transition system, represented by the matrix in (3.2).

Each column in δ corresponds to a state and if there exists a transition $x_i \rightarrow x_j$ then column i in δ contains j . Since G contains the transitions $x_1 \rightarrow x_2$ and $x_1 \rightarrow x_3$, column 1 in δ contains both 2 and 3. Essentially, each column is a list of the target states from all the outgoing transitions of the state corresponding to it. However, since not all states have the same number of outgoing transitions, the columns will not have the same number of elements. This is solved by filling out any remaining places with zeros. In order to create a more efficient implementation, the number of nonzero elements in the columns of δ are stored in δ_n , as

$$\delta_n = \begin{bmatrix} 2 & 1 & 1 & 0 \end{bmatrix}. \quad (3.3)$$

Since a matrix is used to store target states, this representation is called target-matrix representation. An alternative representation, called source-matrix representation, is shown in (3.4), where the matrix γ contains source states rather than target states. To be specific, column i in the matrix contains the source states of all the incoming transitions to state i . A transitions $x_i \rightarrow x_j$ then means that column j contains i . Again, a zero denotes the absence of transitions, and the number of nonzero elements are stored in γ_n .

$$\gamma = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{bmatrix} \quad (3.4)$$

$$\gamma_n = \begin{bmatrix} 0 & 1 & 1 & 2 \end{bmatrix} \quad (3.5)$$

A third possible representation, called full-matrix representation, consists of a binary $N \times N$ matrix and is shown in (3.6). A transition $x_i \rightarrow x_j$ then means that there is a 1 on row i , column j in the matrix.

$$\varepsilon = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (3.6)$$

3.2 Reachability algorithms

The first algorithm, which was proposed by Bengt Lennartsson, uses the target-matrix representation. It is based on the following steps. x represents the states

that have already been reached, d represents newly reached states and o represents states that can be reached from d in one step.

1. Let $x = x_I$ and $d = x_I$
2. Let o be the one-step reachable states from d , by taking the following steps
 - (a) Let J be the set containing the indices of all ones in d
 - (b) For $j \in J$, for $i \in I$, where $I = \{1, 2, \dots, \delta_{n,j}\}$
 - (c) let $o_{\delta_{i,j}} = 1$
3. Let $d = o \wedge \neg x$
4. Let $x = x \vee d$
5. If d contains any ones, repeat from step 2.

This algorithm will be called target-reach. Note how it uses vectorial operations between binary vectors in steps 3 and 4.

The second algorithm uses the source-matrix representation, so it shall be called source-reach. Here, b is used to save the old value of x , so that it keeps looping until the value of x stops changing. The basic idea here is that a particular state x_j is considered reached if it has already been reached or if any of its source states $\gamma_{i,j} | i = [1, \gamma_{n,j}]$ have been.

1. Let $x = x_I$ and $b = x_I$
2. Let $J = 1, 2, \dots, N$
3. For $j \in J$, for $i \in I$, where $I = \{1, 2, \dots, \gamma_{n,j}\}$
4. Let $x_j = x_j \vee x_{\gamma_{i,j}}$
5. If $x \neq b$ then let $b = x$ and repeat from step 2

One main difference between this algorithm and target-reach is that this does not use any vectorial or operation. Instead, it loops through the states one by one.

The third and simplest algorithm uses the full-matrix representation and will be called full-reach. It takes advantage of the fact that a vector matrix multiplication between x and ε produces the one-step-forward reachable states from x .

1. Let $x = x_I$ and $b = x_I$
2. Let $x = x \vee x \cdot \varepsilon$
3. If $x \neq b$ then let $b = x$ and repeat from step 2

The idea here is to exploit the fact that matrix multiplication is very efficient.

3.3 Testing and performance

The algorithms were compared against each other using the transition system G_{SB} shown in Figure 3.2 for different values of N . For a given number N , the transition system has N^2 states and $2N(N - 1)$ transitions, and always uses state 1 as its initial state. From the initial state, all states are reachable in $2(N - 1)$ steps or less.

N	states	target-reach	source-reach	full-reach	source-reach-irn
100	10 000	113	19	18	281
200	40 000	216	52	119	3 600
300	90 000	426	92	540	12 400
400	160 000	893	137	2 190	
500	250 000	1 195	260	4 243	
600	360 000	1 757	391		
700	490 000	2 633	463		
800	640 000	4 635	651		
900	810 000	6 484	798		
1000	1 000 000	9 037	880		

Table 3.1: Execution times in milliseconds from tests in Matlab comparing the three algorithms, as well as a version of source-reach that iterates over states in a random order rather than in ascending order.

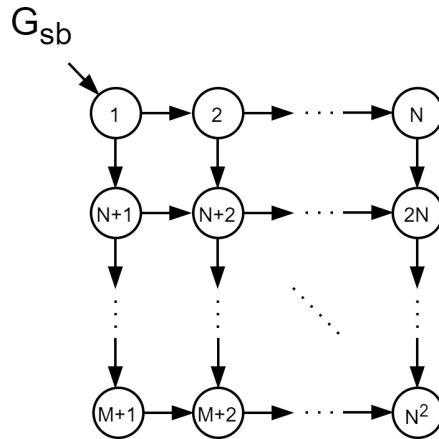


Figure 3.2: A transition system representing two synchronized buffers. In this figure, M is defined as $N^2 - N$.

3.3.1 Testing in Matlab

The tests were run in Matlab using the built in profiler. The results are presented in Table 3.1 and visualized in Figure 3.3. As the results show, source-reach showed great promise initially. Unfortunately, this was due to the specific test example that was used. Consider the two simple transition systems shown in Figure 3.4.

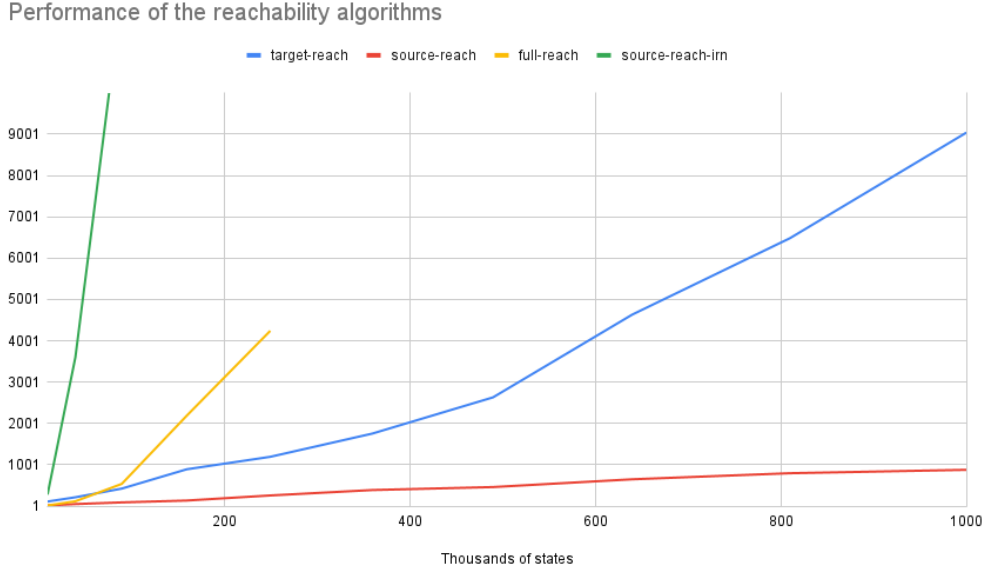


Figure 3.3: Visualization of the test results presented in Table 3.1.

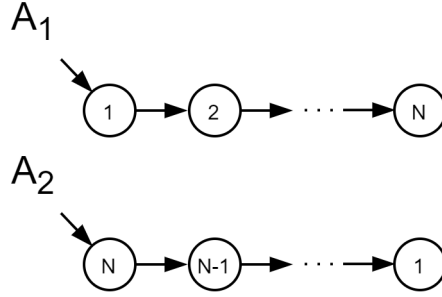


Figure 3.4: Two linear transition systems with states in opposite order.

Since source-reach iterates over the states in ascending order, A_1 is a best-case scenario where all states are found in a single iteration. A_2 , however, is a worst-case scenario where $N - 1$ iterations are required. The example used for the testing, shown in Figure 3.2, is clearly a very favourable case. Because of this, source-reach was modified to iterate over the states in random order. This is what is called source-reach-in-random-order in Table 3.1 and Figure 3.4. The time spent randomizing the order was deducted from the algorithm's run-time. Nevertheless, the modification slowed the algorithm to a crawl.

Interestingly, full-reach proved to be fast for small examples, beating target-reach by a factor of 5 for 10^4 states and almost twice as fast for $4 \cdot 10^4$ states. Unfortunately, it slows down considerably for larger examples as the number of multiplications grows cubically. Additionally, the elements in ε grow quadratically with the number of states in the transition system, which could consume considerable memory. However, there is no data on this since memory use was not measured. Nevertheless, the approach of exploiting the efficiency of matrix multiplication remains attractive for

N	states	target-reach C++	target-reach Matlab
500	250 000	6 600	1 200
1000	1 000 000	46 500	8 600
2000	4 000 000	372 000	73 500

Table 3.2: Execution times in milliseconds from tests comparing target reach in C++ and Matlab.

small examples.

Having produced results in Matlab, target-reach and source-reach were re-implemented in C++. C++ is well known for being a fast language, so it was the natural choice.

3.3.2 Testing in C++

For development in C++, the Visual Studio IDE was used. Measurements of performance were taken using the built-in profiler. Firstly, target-reach and source-reach were implemented so that they followed the Matlab implementation as closely as possible. Code was also written to generate the δ and γ matrices and the corresponding δ_n and γ_n vectors for different values of N . Initial testing of target-reach showed that steps 3 and 4 took about 61% and 21% of the total execution time, respectively. These steps involve vectorial operations \wedge , \neg and \vee on binary vectors. To improve computation time, the datatype used to store vectors was changed from `std::vector<bool>` [15] to `boost::dynamic_bitset<>` [16]. This massively improved performance, allowing the code to handle approximately 10^6 states, whereas it could previously only handle 10^2 whilst still completing in under a minute. More specifically, the test case where $N = 1000$ now took about thirty-five seconds on average. Using the profiler to highlight what areas of the code took the most time, the algorithm was modified several times to improve its execution time. Eventually, it had changed so much that it became equivalent to a breadth-first search (BFS), which I shall call BFS-reach. BFS-reach took only about six seconds to run the test case where $N = 1000$, clearly far superior to reach-target. Three tests were run to compare the differences between implementing target-reach in Matlab and C++. The results are shown in Table 3.2. Target-reach proved to not be very efficient in C++. However, the tests do show a considerable difference between the execution time depending on what language is used. Thus, it was interesting to see if Python (a very popular language) would yield an efficient implementation.

3.3.3 Testing in Python

The development in Python was done using the Pycharm IDE, and the built-in profiler was used for measurements. Target-reach was implemented first and initially used the `List` [17] data structure to store vectors, but this was quickly changed to `numpy.array` [18] to speed up the execution time. Since BFS-reach performed so well in C++, it was implemented next and compared to target-reach. An initial test showed the results presented in Table 3.3.

Once again, we find that BFS-reach is very efficient. Table 3.4 includes the results

N	states	target-reach (List)	target-reach (numpy.array)	BFS-reach
300	90 000	18 000	11 000	200

Table 3.3: Execution times in milliseconds from a test in Python comparing target-reach based on list, target-reach based on numpy.array, as well as BFS-reach.

N	states	BFS-reach
500	250 000	800
1000	1 000 000	3 000
2000	4 000 000	13 200

Table 3.4: Execution times in milliseconds from a test of BFS-reach in python.

from further tests of BFS-reach in python.

3.3.4 Summary

Binary vectors $[x_1 \ x_2 \ \dots \ x_N]$ have been used to represent which states in a state space $X = \{x_1, x_2, \dots, x_N\}$ that have been reached. Transitions have been represented in three ways. In the target-matrix δ , a transition $x_i \rightarrow x_j \in T$ implies that column i in δ contains j . In the source-matrix γ , it implies that column j contains i and in the full-matrix ε , it implies that there is a 1 in position (i, j) . Three algorithms, target-reach, source-reach and full-reach, use these representations, respectively. Importantly, target-reach relies heavily on vectorial operations on binary vectors, whereas source-reach relies on element-wise operations, and full-reach relies on matrix multiplication. The execution times are based on how fast these underlying operations can be made. A fourth algorithm, BFS-reach, was shown to be superior in C++ and Python. Target reach performed best in Matlab for larger examples, surpassed by full-reach for smaller ones.

4

Translation from Petri net to nuXmv

The model checker nuXmv is used for verifying temporal logic specifications on transition systems. It is not intended to be used for Petri nets. Fortunately, bounded Petri nets can be described in the nuXmv input language, but sadly such a description may require a considerable amount of code. This chapter presents a parser, hereby known as PetNet, which solves this issue by translating a compact, user-friendly description of a bounded Petri net into valid nuXmv code. PetNet is implemented as a Matlab script and accepts a text file describing a bounded Petri net that follows a specific format. The text file should be located in the same folder as the script, and running it produces a .smv file, which can, in turn, be run by nuXmv as described in Chapter 2.4.2. The input format is based on the nuXmv input language, as the intended user is likely to be familiar with it. Similarities between the input format and nuXmv code also allowed for a simpler implementation since the script's purpose is to translate from one to another.

4.1 Features

PetNet has four main features, which are outlined here.

1. Any places not initialized by the user will by default be initialized to zero instead of random values.
2. Any places not affected by a transition will by default keep their current values instead of assuming random values.
3. Guards are added, ensuring that the number of tokens in a place cannot be increased beyond its limit, decreased below 0 or set to a value outside this range.
4. Shorthand syntax for common actions.

Out of these, the second is by far the most important as it massively reduces the amount of code the user must write. When writing in nuXmv code, every place must be assigned a new value at every event. Any place that is not assigned a new value will be considered unconstrained and may assume any value inside its domain [4]. In PetNet, however, it is assumed that any place that is not assigned a new value by a given event should keep its current value. Since any given event typically only adds or subtracts tokens from a few places while the majority remains unaffected, this assumption saves the user much time. Much like the second feature assumes that places should keep their number of tokens by default, the first assumes that

the default number of initial tokens is zero.

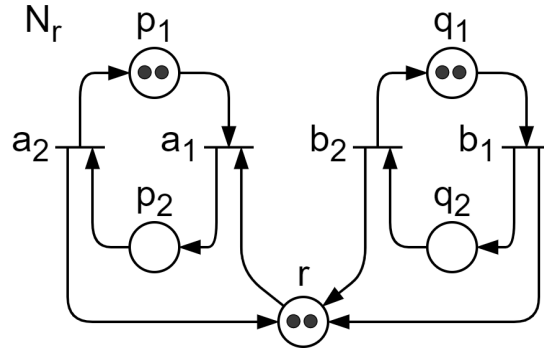
4.2 Syntax

Four keywords are expected to be found in the text, with four more being optional. The necessary keywords are `CONST`, `PLACES`, `INIT`, `TRANS` and the optional ones are `SPEC`, `LTLSPEC`, `INVARSPEC`, `NUXMV`. Each keyword marks the beginning of a section of the code and ends when the next keyword appears, except for `NUXMV` which continues until the end of the file. The reason for this exception will be made clear later on. Comments begin with `--` and continue until the end of the line. Under the `CONST` keyword, a list of declaration of constants is expected, separated by commas and ending with a semicolon. These constants can be used later in the code. Under `PLACES`, a list of places is expected, again separated by commas and ending with a semicolon. Each place is to be directly followed by a colon and an upper limit on the number of tokens that place may contain. The lower limit is always zero. Under `INIT`, the initial number of tokens for each place is declared, comma-separated and ended with a colon. Finally, under `TRANS`, the bounded Petri nets events are written, separated by colons. Each event starts with an event label, followed by a comma and a number of statements. These statements can either be guards that restrict when the event may fire or actions that affect the tokens in the bounded Petri net. The optional keywords `SPEC`, `LTLSPEC`, `INVARSPEC` keywords are used for writing specifications, which follows the same syntax as nuXmv, described in Chapter 2.4.1. Finally, any text between the `NUXMV` keyword and the end of the file is not affected by PetNet. This allows the user to directly write nuXmv code to be included in the output file. The following code segment shows the syntax for the input language of PetNet.

```
-- this is a comment
CONST
    <constant>=<numerical>,<constant>=<numerical> ... ;
PLACES
    <place>:<numerical>,<place>:<numerical> ... ;
INIT
    <place>=<numerical>,<place>=<numerical> ... ;
TRANS
    <label>:<statement><separator> ... <statement>;
    ...
    <label>:<statement><separator> ... <statement>;
SPEC
    <spec>
LTLSPEC
    <ltlspec>
INVARSPEC
    <invarspec>
NUXMV
    <nuxmv>
```

Listing 4.1: Code segment showing the PetNet syntax.

Syntax tag	Description of valid text	Example
<constant>	alphanumeric string beginning with a letter	m1
<place>	alphanumeric string beginning with a letter	p1
<label>	alphanumeric string beginning with a letter	a1
<numerical>	anything that evaluates to a number	2*(m1+3)
<statement>	A <guard> or an <action> tag	p1>2 or p1+
<guard>	<expression><comparator><expression>	p1>2
<action>	<place><operator><numerical>	p1+
<separator>	One of the symbols & ^ -> <->	&
<comparator>	One of the symbols < > = <= >= !=	<
<operator>	One of the symbols + - * / % ' =	+
<spec>	A CTL specification written in the nuXmv	EF p1=0
<ltlspec>	An LTL specification written in the nuXmv	G p1!=0
<invarspec>	An invariant specification written in the nuXmv	p
<nuxmv>	Any valid nuXmv code	SPEC EF p1>1

Table 4.1: Valid text for the syntax tags.**Figure 4.1:** A bounded Petri net where two processes share a mutual resource r .

Valid text for the syntax tags (enclosed by < and >) are shown in in Table 4.1. Most of the symbols for the <separator>, <comparator> and <operator> tags are intuitive. The operator + obviously means addition, for example. The non-trivial ones are ^, meaning xor, %, meaning modulus, and '=, meaning next. The next operator '= is used to directly assign a specific value to the next value of a place. Assuming p is a place with the limit m and N is a numerical, Table 4.2 shows how the shorthand syntax for actions, constants and places are translated into nuXmv code. The so-called next action $p'=N$ has one small exception that differs from what is shown in the Table, namely that the guard $0 \leq N \ \& \ N \leq m$ is skipped if the value of the numerical N is independent of the number of tokens in the places. So if N is a constant number such as 5 or $2*m1$, then it is the users responsibility to make sure that this lies within the domain of p . If, however, N is something like $p2+1$, then the guards are added. The reasoning behind this is that trying to assign a number of tokens to a place that is always outside of its domain would be a model error, which is easier to debug without the guards as they prevent nuXmv from generating errors.

Meaning	PetNet	nuXmv
increment	p^+	$\text{next}(p)=p+1 \ \& \ p+1 \leq m$
decrement	p^-	$\text{next}(p)=p-1 \ \& \ p-1 \geq 0$
increase by N	$p+N$	$\text{next}(p)=p+N \ \& \ p+N \leq m$
decrease by N	$p-N$	$\text{next}(p)=p-N \ \& \ p-N \geq 0$
multiply by N	$p*N$	$\text{next}(p)=p*N \ \& \ p*N \leq m$
divide by N	p/N	$\text{next}(p)=p/N \ \& \ p/N \geq 0$
modulus N	$p\%N$	$\text{next}(p)=p\%N \ \& \ p\%N \geq 0$
next is N	$p'=N$	$\text{next}(p)=N \ \& \ 0 \leq N \ \& \ N \leq m$
constant def.	$m=N$	$m:=N$
place def.	$p:m$	$p:0..m$

Table 4.2: Shorthand syntax for actions and definitions of constants and places. Here, p is a place with the limit m and N is a numerical.

To clarify the syntax, the bounded Petri net N_r , shown in Figure 4.1, along with two specifications, is described in PetNet code in Listing 4.2. The nuXmv code that PetNet generates from this is shown as Listing 4.3.

Examples are all well and good but do not prove that any bounded Petri net as defined in Chapter 2.1.3 may be expressed. Thus it is now shown how such a definition may be translated into PetNet code.

```

CONST
  m=2;
PLACES
  p1:m, p2:m, q1:m, q2:m, r:2;
INIT
  p1=m, q1=m, r=1;
TRANS
  a1: p1- & r- & p2+;
  a2: p1+ & r+ & p2-;
  b1: q1- & r- & q2+;
  b2: q1+ & r+ & q2-;
SPEC -- The resource r always returned eventually
  AG EF (r=1)
LTLSPEC -- Mutual exclusion between p2 and q2
  G !(p2=1 & q2=1)

```

Listing 4.2: PetNet code describing the bounded Petri net N_r shown in Figure 4.1

```

MODULE main

  DEFINE
    m=2;

  VAR
    p1:0..m; p2:0..m; q1:0..m; q2:0..m; r:0..2;

  IVAR
    event:{a1, a2, b1, b2};

  INIT
    p1=m & q1=m & r=1 & p2=0 & q2=0;

  TRANS
    (event=a1 & next(p1)=p1 -1 & p1 -1>=0 & next(r)=r -1 & r
    -1>=0 & next(p2)=p2+1 & p2+1<=m & next(q1)=q1 & next(q2)=q2) |
    (event=a2 & next(p1)=p1+1 & p1+1<=m & next(r)=r+1 & r+1<=2 &
    next(p2)=p2 -1 & p2 -1>=0 & next(q1)=q1 & next(q2)=q2) |
    (event=b1 & next(q1)=q1 -1 & q1 -1>=0 & next(r)=r -1 & r
    -1>=0 & next(q2)=q2+1 & q2+1<=m & next(p1)=p1 & next(p2)=p2) |
    (event=b2 & next(q1)=q1+1 & q1+1<=m & next(r)=r+1 & r+1<=2 &
    next(q2)=q2 -1 & q2 -1>=0 & next(p1)=p1 & next(p2)=p2) ;

  SPEC AG EF (r=1)
  LTLSPEC G !(p2=1 & q2=1)

```

Listing 4.3: nuXmv code describing the bounded Petri net N_r shown in Figure 4.1

4.3 Definition to PetNet code

Consider a bounded Petri net

$$N_B = \langle P, T, I, D, M, E, L \rangle \quad (4.1)$$

with n places p_i for $i \in [1, n]$ and k events e_j for $j \in [1, k]$. Each place $p_i \in P$ has the initial number of tokens $m_i \in M$ and upper limit $l_i \in L$. Each transition $t_j \in T$ corresponds to row j in the incidence matrix I and decadence matrix D , which are both of dimension $n \times k$. Each transition $t_j \in T$, corresponds to an event label $e_j \in E$. If the condition

$$(I_{i,j} = 0 \vee D_{i,j} = 0) \forall i \in \{1, 2, \dots, n\}, j \in \{1, 2, \dots, m\} \quad (4.2)$$

holds, then there is no event that both takes and gives tokens to the same place and the Petri net is translated into PetNet code as follows, where $A = I - D$

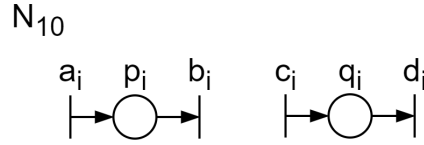


Figure 4.2: Petri net example where $i \in [0, 9]$.

```

CONST

PLACES
    p1:l1, p2:l2, ... pn:ln;
INIT
    p1=m1, p2=m2, ... pn=mn
TRANS
    e1:p1+A11 & p2+A21 & ... & pn+An1;
    e2:p1+A12 & p2+A22 & ... & pn+An2;
    ...
    ek:p1+A1k & p2+A2k & ... & pn+Amk;

```

Listing 4.4: PetNet code for the bounded Petri net defined in (4.1).

Now consider that (4.2) does not hold for some (i, j) . Then the following code must be appended to the line in the TRANS block that corresponds to e_i .

```
pj-Dij >= 0
```

Listing 4.5: PetNet guard statement ensuring that an event only fires if there are sufficient tokens available.

This will ensure that the event only fires if p_j contains a sufficient number of tokens. This concludes how a bounded Petri net is described as PetNet code. However, one can note that the full syntax of PetNet also allows for other behaviour, such as an event setting the number of tokens in a place to a specific value or multiplying the number of tokens in a place with some number. Of course, none of this is interesting if the syntax for PetNet is not more convenient than that of nuXmv. Therefore, the translator's performance shall now be considered. The performance will be measured in terms of the number of characters necessary to describe a given Petri net. The fewer the characters, the better. Of course, this must be compared to the number of characters that the user would have had to write if they were not using PetNet. Thus, we will use the relative difference between the number of characters needed to describe Petri nets as our performance measure.

4.4 Performance

Consider the Petri net N_{10} , shown in Figure 4.2, where $i \in [0, 9]$. It consists of 20 places, each with two corresponding transitions that increment and decrement the number of tokens in them. Each place initially has zero tokens, with an upper limit of ten. This example has been chosen since it has many simple transitions spread across a large number of places. The PetNet code describing N_{10} is shown below in

Figure 4.3: nuXmv description of the Petri net example from Listing 4.6.

Listing 4.6. It is 530 characters long and gives a compact and readable description of the Petri net. More importantly, it can be written quickly.

```

CONST
    m=10;
PLACES
    p0:m, p1:m, p2:m, p3:m, p4:m,
    p5:m, p6:m, p7:m, p8:m, p9:m,

    q0:m, q1:m, q2:m, q3:m, q4:m,
    q5:m, q6:m, q7:m, q8:m, q9:m;
INIT
TRANS
    a0:p0+; a1:p1+; a2:p2+; a3:p3+; a4:p4+;
    a5:p5+; a6:p6+; a7:p7+; a8:p8+; a9:p9+;

    b0:p0-; b1:p1-; b2:p2-; b3:p3-; b4:p4-;
    b5:p5-; b6:p6-; b7:p7-; b8:p8-; b9:p9-;

    c0:q0+; c1:q1+; c2:q2+; c3:q3+; c4:q4+;
    c5:q5+; c6:q6+; c7:q7+; c8:q8+; c9:q9+;

    d0:q0-; d1:q1-; d2:q2-; d3:q3-; d4:q4-;
    d5:q5-; d6:q6-; d7:q7-; d8:q8-; d9:q9-;

```

Listing 4.6: PetNet code representing the Petri net shown in figure 4.2.

The equivalent description of N_{10} in nuXmv code has 12 806 characters and is shown in Figure 4.3. This gives us a relative difference in characters of $\frac{12806-530}{12806} \approx 96\%$. Of course, this example is designed to highlight the strengths of PetNet. As one can see in Figure 4.3, most characters lie under the **TRANS** keyword. Let us thus consider an arbitrary Petri net, containing N places and M transitions. Each of the N places must be assigned some value in each of the M transitions. Thus, there must be $N \cdot M$ assignments in total. N_{10} contains 20 places and 40 transitions, creating 800 assignments. In PetNet, however, there is no need to write these assignments.

4.5 Summary

While bounded Petri nets can be described in the nuXmv input language, it often requires much code. A parser, known as PetNet and implemented as a Matlab script, accepts a brief description of a Petri net as a text file and translates it into valid nuXmv code. The relative difference between the number of characters needed for the PetNet description and the nuXmv description has been shown to be large, especially for Petri nets with many states and transitions. If most transitions do not affect most places, as is often the case, this difference widens. PetNet effectively extends nuXmv to be used not only for transition systems but also for Petri nets.

5

Comparing nuXmv based model checking with incremental abstraction

Incremental abstraction and nuXmv have been introduced as tools for temporal logic verification. However, both of these apply only to transition systems, not Petri nets. Fortunately, a Petri net may be translated into a transition system, either by generating its reachability graph or by replacing places with buffer transition systems as shown in Chapter 2.1.5. Thus, incremental abstraction may be applied if a transition system is generated from the Petri net. As for nuXmv, Chapter 4.3 shows how PetNet can be used to convert a description of a Petri net into valid nuXmv code, representing a transition system with the same behaviour. These two methods have been compared against each other in [7], and that comparison will now be explained and discussed in the context of this thesis. The test example is a bounded, modular Petri net.

5.1 Model used for testing

The temporal logic verification methods will be compared against each other on the bounded, modular Petri net shown in Figure 5.1. It consists of several straight sequences of places of finite capacity and three shared resources R_1 , R_2 and R_3 . This models a classic dual production line. The tokens then model products, and the places model machines that process them individually separated by buffers of size m . The synchronisation of these Petri nets is shown in Figure 5.2. Since PN is bounded, it has a finite state space. It does not, however, have finite paths lengths, meaning that execution could continue infinitely. The a transitions could, for example, repeatedly fire in ascending order infinitely, first a_0 , then a_1 and so on up until a_{13} , after which a_0 fires again and the sequence repeats itself. One may remark that this behaviour is not fair since the b transitions then never fire. One simple way to achieve fairness here is by synchronizing with the transition system F , which forces the events a_2 and b_2 to alternate. F is shown in Figure 5.3 and the fair Petri net PN_f is defined as $PN_f = PN || F$.

Infinite path lengths are desirable when testing fairness, but a final state can also be of interest. This can be achieved by limiting the number of tokens that can enter and exit the Petri net. For PN , the entering tokens can be limited by restricting the number of times a_0 and b_0 can fire. Similarly, the exiting tokens can be limited where

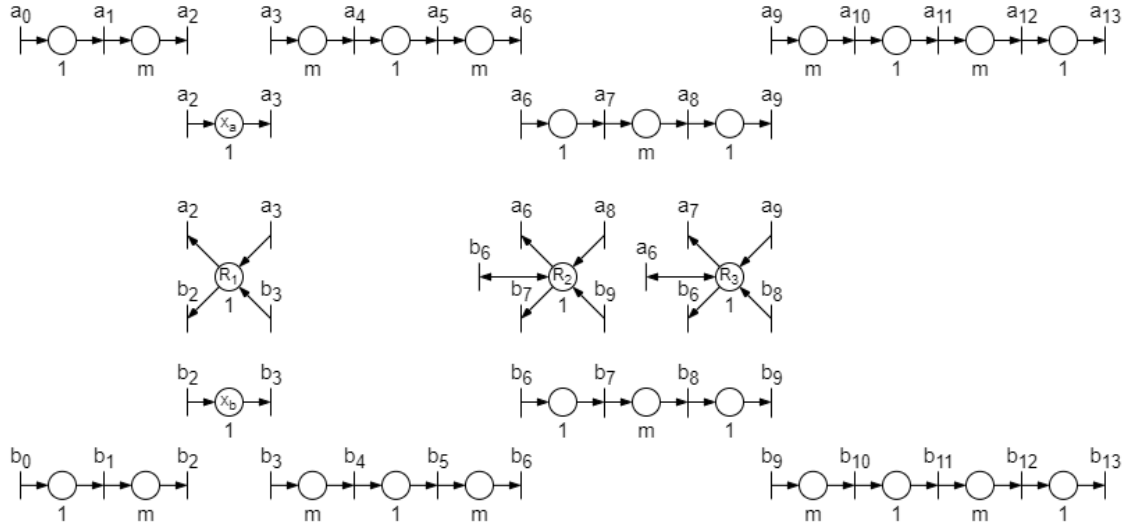


Figure 5.1: The modular Petri net PN , before synchronisation.

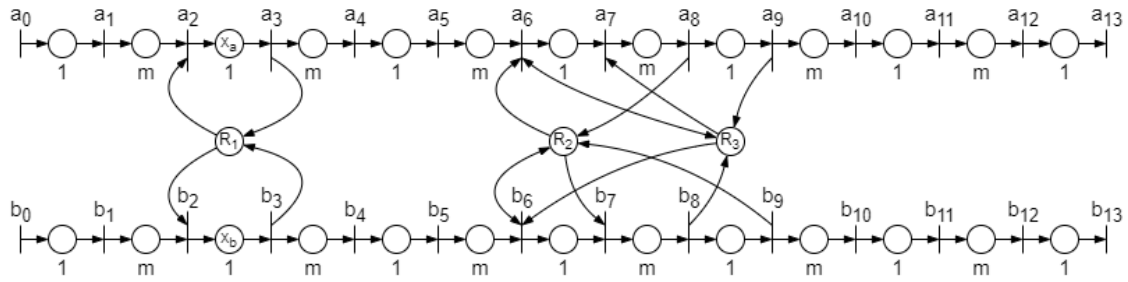


Figure 5.2: The modular Petri net PN , after synchronisation.

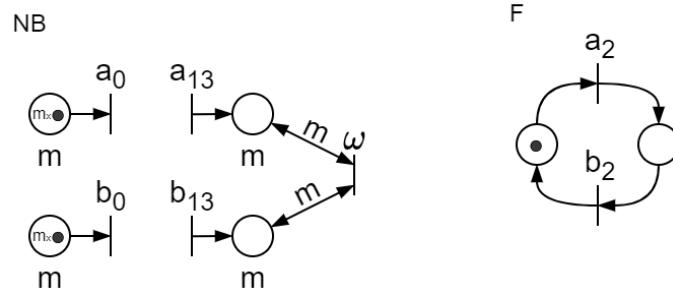


Figure 5.3: The Petri net NB and F . They are synchronised with PN to create PN_{nb} and PN_f , respectively.

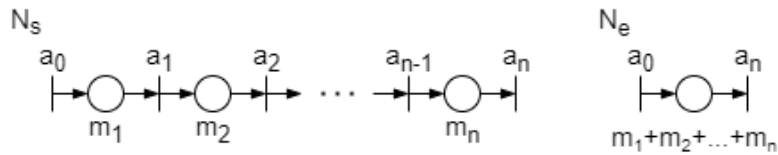


Figure 5.4: A Petri net consisting of a straight sequence of places as well as the single place Petri net it may be abstracted to if all events but a_0 and a_n are local.

the number of times a_{13} and b_{13} can fire is restricted. This gives initial and final states. The initial state is when there are m tokens in the places before a_0 and b_0 , as shown in NB in figure 5.3. The final state is when there are m tokens the places after a_{13} and a_{14} , also shown in NB [7]. However, nuXmv expects infinite path lengths [14], so the ω self-loop is added to the final state. The ω event, may only fire in the final state and does not change the state. The only effect it has is that it changes the final state from a deadlock to a livelock and thus allows nuXmv to handle our model. Thus, the non-blocking Petri net PN_{nb} is obtained as $PN_{nb} = PN || NB$, where NB is shown in Figure 5.3.

So there are now two versions of the original Petri net, namely PN_f and PN_{nb} , which are used to test fairness and non-blocking, respectively. These will be used to compare incremental abstraction and nuXmv as methods for temporal logic verification. However, another method, known as analytical abstraction, will first be introduced. The goal of this method will not be to perform verification but to abstract PN by hand before applying any verification method.

5.2 Analytical abstraction

Consider the Petri net N_s comprising a sequence of places, as seen in Figure 5.4. If it is assumed that the events a_1, \dots, a_{n-1} are local, then N_s can be abstracted to N_e , also shown in Figure 5.4. The abstracted Petri net consists of a single place with a capacity equal to the sum of all capacities in N_s . Suppose now that either a_0 or a_n is also a local transition and that the temporal logic specification that is to be validated does not depend on the number of tokens in the place between them. Then, the Petri net can be abstracted completely and removed from the model [7]. To understand this, consider the unsynchronized PN model shown in Figure 5.1. If the idea of abstracting sequences of places into a single place with extended capacity is applied, the abstracted Petri net shown in Figure 5.5 is obtained. As one can see, six sequences were replaced by single places. Note that when doing this, the specifications that one wishes to validate must be respected. If a specification depends on a certain transition or the number of tokens in a certain place, then that transition or place cannot be abstracted away. In the case of PN_{nb} , no further abstractions are possible, and one ends up with the Petri net shown in Figure 5.6. However, the fairness condition $\varphi_f = \Box \diamond x_a \wedge \Box \diamond x_b$ on PN_f does not depend on the number of tokens in the first two places or the final place. Thus, the model PN_f may be abstracted further, resulting in the Petri net shown in Figure 5.7.

Further abstractions are possible for both PN_{nb}^A and PN_f^A , but more difficult to make. However, a significant abstraction has already been made, saving computation time when more sophisticated methods are applied later. This is the point of analytical abstraction. By making simple yet powerful abstractions by hand, the time needed to run temporal verification methods such as incremental abstraction and nuXmv is greatly reduced.

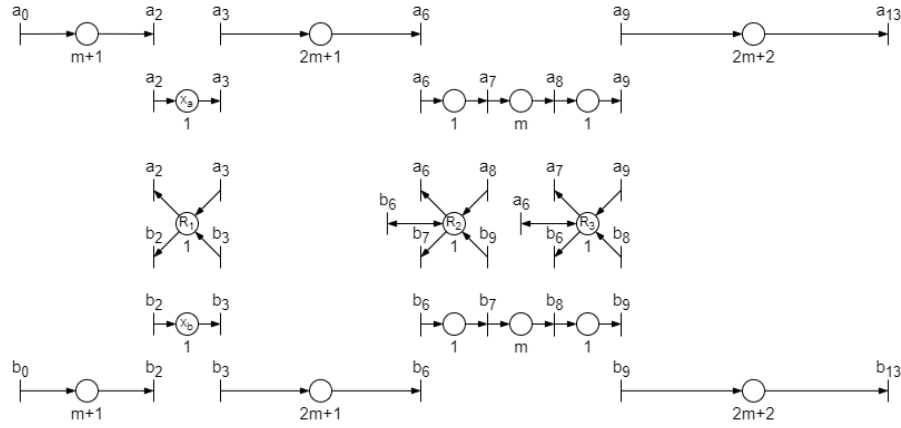


Figure 5.5: The abstracted Petri net PN^A .

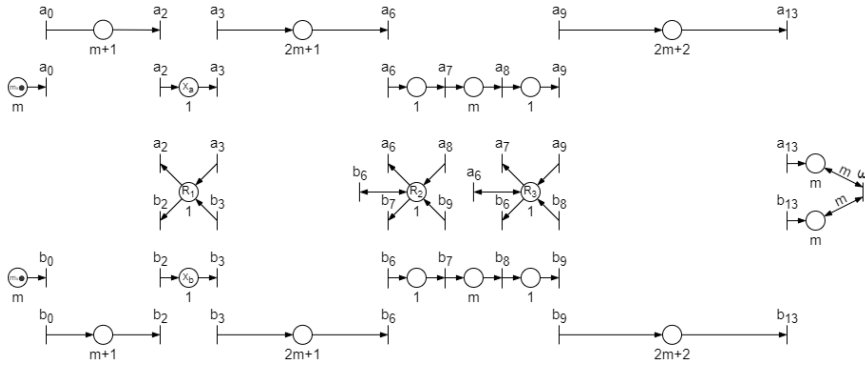


Figure 5.6: The abstracted nonblocking Petri net PN_{nb}^A .

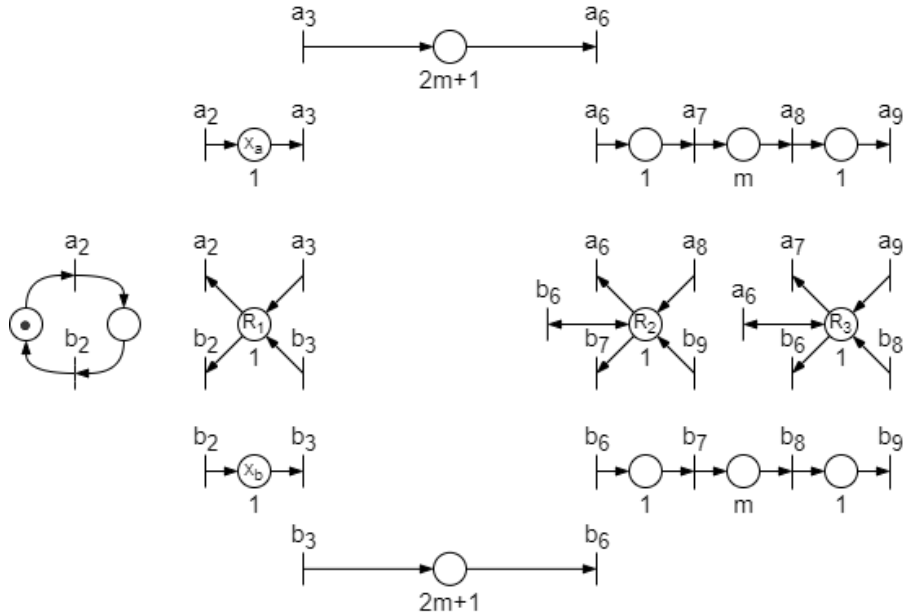


Figure 5.7: The abstracted fair Petri net PN_f^A .

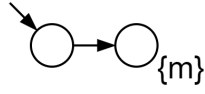


Figure 5.8: The fully abstracted nonblocking Petri net PN_{nb}^{FA} .

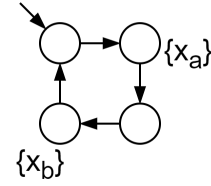


Figure 5.9: The fully abstracted fair Petri net PN_f^{FA} .

5.3 Incremental abstraction

Incremental abstraction is now applied on PN_{nb} , PN_{nb}^A , PN_f and PN_f^A . Applying it on PN_{nb} or PN_{nb}^A results in the simple transition system PN_{nb}^{FA} shown in Figure 5.8, which one can see is trivially non-blocking since it only has a single transition that immediately takes the transition system into the marked state with the state label m .

Applying incremental abstraction to PN_f or PN_f^A also results in a very simple transition system, namely PN_f^{FA} , shown in Figure 5.9. Again, it is trivial to evaluate fairness on this reduced transition system, as the propositions x_a and x_b obviously become true infinitely often. The beauty here is that there is no need to verify that PN_f is fair or that PN_{nb} non-blocking. It is not even necessary to generate PN_f (by synchronizing its subsystems) because PN_f is fair if and only if PN_f^{FA} is. Similarly, PN_{nb} is non-blocking if and only if PN_{nb}^{FA} is. Since the verification of the abstracted models is so easy, the concern now lies only with how quickly a model can be abstracted. The results for this are shown in Chapter 5.5.

5.4 Modelling in PetNet and nuXmv analysis

In order for us to use nuXmv to verify our temporal logic specifications, PN_{nb} , PN_{nb}^A , PN_f and PN_f^A must first be modelled in PetNet. These models are shown in code-listing 5.1, 5.2, 5.3 and 5.4, respectively. The number of characters these models consist of, as well as the number of characters in the corresponding nuXmv models and the relative difference in characters are all shown in Table ???. Note that all of these models are using $m = 2$, but the difference in characters when changing m is completely negligible. m determines the difficulty of the verification step, not the modelling.

Running the generated nuXmv code while measuring the time it takes to validate gives the absolute computation times. Of course, the absolute computation times do not say much since they heavily depend on the machine the code runs on. What is interesting is the relative difference in computation time between different models and between this approach and incremental abstraction. For this reason, the results are presented in Chapter 5.5.

```

CONST
  m:=2;
PLACES
  p0:m, p1:1, p2:m, p3:1, p4:m, p5:1, p6:m, p7:1, p8:m, p9:1,
  p10:m, p11:1, p12:m, p13:1, p14:m,

  q0:m, q1:1, q2:m, q3:1, q4:m, q5:1, q6:m, q7:1, q8:m, q9:1,
  q10:m, q11:1, q12:m, q13:1, q14:m,

  R1:1, R2:1, R3:1;
INIT
  p0=m, q0=m, R1=1, R2=1, R3=1;
TRANS
  a0: p0- & p1+;          b0: q0- & q1+;
  a1: p1- & p2+;          b1: q1- & q2+;
  a2: p2- & p3+ & R1-;    b2: q2- & q3+ & R1-;
  a3: p3- & p4+ & R1+;    b3: q3- & q4+ & R1+;
  a4: p4- & p5+;          b4: q4- & q5+;
  a5: p5- & p6+;          b5: q5- & q6+;
  a6: p6- & p7+ & R2- & R3>0; b6: q6- & q7+ & R3- & R2>0;
  a7: p7- & p8+ & R3-;    b7: q7- & q8+ & R2-;
  a8: p8- & p9+ & R2+;    b8: q8- & q9+ & R3+;
  a9: p9- & p10+ & R3+;   b9: q9- & q10+ & R2+;
  a10: p10- & p11+;       b10: q10- & q11+;
  a11: p11- & p12+;       b11: q11- & q12+;
  a12: p12- & p13+;       b12: q12- & q13+;
  a13: p13- & p14+;       b13: q13- & q14+;
  w: p14=m & q14=m;
SPEC
  AG EF (p14 = m & q14 = m);

```

Listing 5.1: PetNet model of PN_{nb} .

```

CONST
  m:=2;
PLACES
  p0:m, p2:m, p3:1, p6:m, p7:1, p8:m, p9:1, p13:1, p14:m,
  q0:m, q2:m, q3:1, q6:m, q7:1, q8:m, q9:1, q13:1, q14:m,
  R1:1, R2:1, R3:1;
INIT
  p0=m, q0=m, R1=1, R2=1, R3=1;
TRANS
  a0: p0- & p2+;          b0: q0- & q2+;
  a2: p2- & p3+ & R1-;    b2: q2- & q3+ & R1-;
  a3: p3- & p6+ & R1+;    b3: q3- & q6+ & R1+;
  a6: p6- & p7+ & R2- & R3>0; b6: q6- & q7+ & R3- & R2>0;
  a7: p7- & p8+ & R3-;    b7: q7- & q8+ & R2-;
  a8: p8- & p9+ & R2+;    b8: q8- & q9+ & R3+;
  a9: p9- & p13+ & R3+;   b9: q9- & q13+ & R2+;
  a13: p13- & p14+;       b13: q13- & q14+;
  w: p14=m & q14=m;
SPEC
  AG EF (p14 = m & q14 = m);

```

Listing 5.2: PetNet model of the analytically abstracted PN_{nb}^A .

```

CONST
  m:=2;
PLACES
  p1:1, p2:m, p3:1,  p4:m, p5:1,  p6:m, p7:1, p8:m, p9:1,
  p10:m, p11:1, p12:m, p13:1,

  q1:1, q2:m, q3:1,  q4:m, q5:1,  q6:m, q7:1, q8:m, q9:1,
  q10:m, q11:1, q12:m, q13:1,

  R1:1, R2:1, R3:1, R4:1;
INIT
  R1=1, R2=1, R3=1;
TRANS
  a0: p1+;                b0: q1+;
  a1: p1- & p2+;          b1: q1- & q2+;
  a2: p2- & p3+ & R1- & R4+; b2: q2- & q3+ & R1- & R4-;
  a3: p3- & p4+ & R1+;    b3: q3- & q4+ & R1+;
  a4: p4- & p5+;          b4: q4- & q5+;
  a5: p5- & p6+;          b5: q5- & q6+;
  a6: p6- & p7+ & R2- & R3>0; b6: q6- & q7+ & R3- & R2>0;
  a7: p7- & p8+ & R3-;    b7: q7- & q8+ & R2-;
  a8: p8- & p9+ & R2+;    b8: q8- & q9+ & R3+;
  a9: p9- & p10+ & R3+;   b9: q9- & q10+ & R2+;
  a10: p10- & p11+;       b10: q10- & q11+;
  a11: p11- & p12+;       b11: q11- & q12+;
  a12: p12- & p13+;       b12: q12- & q13+;
  a13: p13-;              b13: q13-;
LTLSPEC
  (G F p3=1) & (G F q3=1);

```

Listing 5.3: PetNet model of PN_f .

```

CONST
  m:=2;
PLACES
  p3:1, p6:m, p7:1, p8:m, p9:1,
  q3:1, q6:m, q7:1, q8:m, q9:1,
  R1:1, R2:1, R3:1, R4:1;
INIT
  R1=1, R2=1, R3=1;
TRANS
  a2: p3+ & R1- & R4+;    b2: q3+ & R1- & R4-;
  a3: p3- & p6+ & R1+;    b3: q3- & q6+ & R1+;
  a6: p6- & p7+ & R2- & R3>0; b6: q6- & q7+ & R3- & R2>0;
  a7: p7- & p8+ & R3-;    b7: q7- & q8+ & R2-;
  a8: p8- & p9+ & R2+;    b8: q8- & q9+ & R3+;
  a9: p9- & R3+;          b9: q9- & R2+;
LTLSPEC
  (G F p3=1) & (G F q3=1);

```

Listing 5.4: PetNet model of the analytically abstracted PN_f^A .

Model	Characters (PetNet)	Characters (nuXmv)	Reduction
PN_{nb}	976	16070	94%
PN_{nb}^A	923	6470	86%
PN_f	626	14189	96%
PN_f^A	456	3325	86%

Table 5.1: Number of characters used to represent various models in PetNet and nuXmv.

m	2	6	10
PN_f	T.O.	T.O.	T.O.
PN_f^A	0.21	1.43	4.95
PN_{nb}	0.23	41.5	575
PN_{nb}^A	0.05	1.28	7.66

Table 5.2: Test data for using nuXmv to validate the nonblocking and fairness properties.

m	2	6	10
PN_f	0.8	0.9	1.0
PN_{nb}	0.8	1.5	2.9

Table 5.3: Test data for using incremental abstraction to validate the nonblocking and fairness properties.

5.5 Results

The results from using nuXmv and incremental abstractions to verify the nonblocking and fairness properties are presented in Table 5.2 and 5.3, respectively, and taken from [7]. Here, T.O. stands for time out, which means that it had not finished within the 600-second limit. The analytically abstracted models PN_f^A and PN_{nb}^A are only verified using nuXmv. There are no issues with applying incremental abstraction to an already abstracted model, but that has not been done in this case. As the data shows, incremental abstraction outperforms nuXmv, even when nuXmv is given the substantial benefits of analytical abstraction. When nuXmv must work with the unabstracted models PN_f and PN_{nb} , it takes considerably longer for verifying non-blocking and fails altogether on verifying fairness.

5.6 Summary

This chapter compares nuXmv and incremental abstraction as methods for temporal logic verification. A bounded modular Petri net PN , shown in Figure 5.2 is used as a test example, where fairness and non-blocking are to be verified. Fairness requires that certain places receive tokens infinitely often, meaning that execution must be able to continue infinitely. However, it is also interesting to study models that have a final state and for such a model non-blocking is tested. Thus the non-blocking model PN_{nb} and the fair model PN_f are created. Analytical abstraction is introduced, which involves making simple yet significant abstractions to the models before more involved methods are applied. The results of applying incremental abstraction are presented. The models as written in PetNet are shown. Finally, the computation times between the different methods with and without analytical abstraction are

presented, showing that incremental abstraction is a powerful tool, both when used by itself and as the base for analytical abstraction.

6

Conclusion

This chapter summarises the conclusions that can be drawn from this thesis with respect to the research questions presented in Chapter 1.2. Because the questions touch on separate topics, this chapter is structured to answer the questions separately.

6.1 Efficient reachability

One main goal of this thesis has been to investigate ways to implement a reachability algorithm efficiently. For this, a Matlab implementation based on repeatedly performing bitwise operations between boolean vectors was used as a starting point. Two alternative algorithms were developed, and the algorithms were tested against each other. Implementations were also made in both C++ and Python where comparisons were also made against a breadth-first search. Based on this work, the following conclusions were made.

- Matlabs efficient matrix multiplication may be exploited to achieve a very efficient way to find the reachable states of transition systems with around 10^4 states or less. However, this method slows down considerably for larger transition systems and has poor memory complexity.
- In C++, the `boost::dynamic_bitset<>` data-structure is significantly faster than `std::vector<bool>` at performing bitwise boolean operations. However, such operations are still significantly faster in Matlab.
- In Python, the `numpy.array` data structure is significantly faster than list comprehension at performing bitwise boolean operations. However, such operations are still faster in Matlab.
- Because bitwise boolean operations can be made faster in Matlab than in C++ or Python, an algorithm based on such operations (such as target-reach) will be more efficient in Matlab than in C++ or Python.
- In C++ and Python, a breadth-first search is an efficient way of implementing a reachability algorithm.

6.2 PetNet

This thesis has presented a parser that converts a brief description of a bounded Petri net into nuXmv code. The parser, known as PetNet, accepts definitions of constants as well as places, their limits and an initial number of tokens. Furthermore, the user must describe the transitions and their event labels, guards and actions, and any LTL

and CTL specifications that are to be validated. The primary differences between the PetNet input code and the nuXmv code generated are given by the following four main features.

1. Any places not initialized by the user will by default be initialized to zero instead of random values.
2. Any places not affected by a transition will by default keep their current values instead of assuming random values.
3. Guards are added, ensuring that the number of tokens in a place cannot be increased beyond its limit, decreased below 0 or set to a value outside this range.
4. Shorthand syntax for common actions.

Of these, the second feature is responsible for the bulk of the reduction. This is because, in nuXmv, one must specify the new number of tokens in each place for each transition. If the new number of tokens in a certain place is left unspecified for a certain transition, that transition may assign any number of tokens to that place. Most transitions do not affect most places, so a great deal of assignments that state that the new number is the same as the old one must be included in nuXmv. Removing the need to write these assignments thus eliminates a large part of the code one has to write. The result is that the parsers performance (measured as the relative difference between the number of characters in the PetNet code and equivalent nuXmv code) is very high, especially for Petri nets with many places and transitions.

6.3 Incremental abstraction and nuXmv

This thesis has been concerned with efficient temporal logic verification. It has compared two methods for this, namely incremental abstraction and nuXmv. Fairness and non-blocking properties were verified on a test example consisting of a bounded modular Petri net. Fairness requires that a certain condition is repeated indefinitely while non-blocking requires that a marked state is reached. One fair and one non-blocking version of the Petri net was created to test these properties separately. The results showed incremental abstraction to be the superior method. Additionally, Analytical abstraction is introduced. Based on the same principles as incremental abstraction, it is a way of making simple yet powerful abstractions to a model before we apply more comprehensive methods. Test results showed that analytical abstraction significantly reduced the computation time needed for validation.

Bibliography

- [1] Lennartson B., Introduction to Discrete Event Systems Lecture Notes Chapter 1-8
- [2] Christos G. Cassandras, Stéphane Lafortune (2008) Introduction to Discrete Event Systems Second Edition
- [3] Wimmer R., Herbstritt M., Hermanns H., Strampp K., Becker B. (2006) Sigref – A Symbolic Bisimulation Tool Box. In: Graf S., Zhang W. (eds) Automated Technology for Verification and Analysis. ATVA 2006. Lecture Notes in Computer Science, vol 4218. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11901914_35
- [4] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, M. Roveri (2015) NuSMV 2.6 Tutorial
- [5] Lennartson B., Liang X., Noori-Hossein M. (2020) Efficient Temporal Logic Verification by Incremental Abstraction
- [6] C. A. R. Hoare, Communicating Sequential Processes, ser. Series in Computer Science. ACM, Aug. 1978, vol. 21.
- [7] Lennartson B., (2021) Incremental Abstraction - An Analytical and Algorithmic Perspective on Petri Net Reduction*
- [8] Lennartson B., Introduction to Discrete Event Systems Lecture Notes Chapter 9
- [9] C. Baier and J. P. Katoen, Principles of Model Checking. Cambridge, MA: The MIT Press, 2008
- [10] Lennartson B., Introduction to Discrete Event Systems Lecture Notes - Mu calculus
- [11] Lennartson B., Introduction to Discrete Event Systems Lecture Notes - Bisimulation and Compositional Reduction
- [12] R. Cavada, A. Cimatti, R. Sebastiani M. Dorigatti, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, (2015) The nuSMV Symbolic Model Checker <https://nusmv.fbk.eu/>
- [13] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, (2014) The nuXmv Symbolic Model Checker <https://nuxmv.fbk.eu/>
- [14] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta (2019) nuXmv 2.0.0 User Manual
- [15] Unknown C++ enthusiast (2021). Available: https://en.cppreference.com/w/cpp/container/vector_bool

- [16] The C++ Standards Committee Library Working Group (2006). Available: https://www.boost.org/doc/libs/1_76_0/libs/dynamic_bitset/dynamic_bits_et.html
- [17] The Python Software Foundation (2021). Available: <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>
- [18] The NumPy community (2021). Available: <https://numpy.org/doc/stable/reference/generated/numpy.array.html>
- [19] B. Lennartson, F. Basile, S. Miremadi, Z. Fei, M. Noori-Hosseini, M. Fabian, and K. Åkesson, “Supervisory control for state-vector transition models - A unified approach,” *IEEE Transaction on Automation Science and Engineering*, vol. 11, no. 1, 2014.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se



CHALMERS
UNIVERSITY OF TECHNOLOGY