

Specification and Verification of Network  
Algorithms Using Temporal Logic

by

Ra'ed Bani Abdelrahman

A Doctoral Thesis

Submitted in partial fulfilment  
of the requirements for the award of

Doctor of Philosophy  
of  
Loughborough University

12 December 2018

Copyright 2018 Ra'ed Bani Abdelrahman

# Abstract

In software engineering, formal methods are mathematical-based techniques that are used in the specification, development and verification of algorithms and programs in order to provide reliability and robustness of systems. One of the most difficult challenges for software engineering is to tackle the complexity of algorithms and software found in concurrent systems. Networked systems have come to prominence in many aspects of modern life, and therefore software engineering techniques for treating concurrency in such systems has acquired a particular importance. Algorithms in the software of concurrent systems are used to accomplish certain tasks which need to comply with the properties required of the system as a whole. These properties can be broadly subdivided into ‘safety properties’, where the requirement is ‘nothing bad will happen’, and ‘liveness properties’, where the requirement is that ‘something good will happen’. As such, specifying network algorithms and their safety and liveness properties through formal methods is the aim of the research presented in this thesis. Since temporal logic has proved to be a successful technique in formal methods, which have various practical applications due to the availability of powerful model-checking tools such as the NuSMV model checker, we will investigate the specification and verification of network algorithms using temporal logic and model checking. In the first part of the thesis, we specify and verify safety properties for network algorithms. We will use temporal logic to prove the safety property of data consistency or serializability for a model of the execution of an unbounded number of concurrent transactions over time, which could represent software schedulers for an unknown number of transactions being present in a network. In the second part of the thesis, we will specify and verify the liveness properties of networked flooding algorithms.

Considering the above in more detail, the first part of this thesis specifies a model of the execution of an unbounded number of concurrent transactions over time in propositional Linear Temporal Logic (LTL) in order to prove serializability. This is made possible by assuming that data items are ordered and that the transactions accessing these data items respects this order, as then there is a bound on the number of transactions that need to be considered to prove serializability. In particular, we make use of recent work which places such bounds on the number

of transactions needed when data items are accessed in order, but do not have to be accessed contiguously, i.e., there may be ‘gaps’ in the data items being accessed by individual transactions. Our aim is to specify the concurrent modification of data held on routers in a network as a transactional model. The correctness of the routing protocol and ensuring safety and reliability then corresponds to the serializability of the transactions. We specify an example of routing in a network and the corresponding serializability condition in LTL. This is then coded up in the NuSMV model checker and proofs are performed. The novelty of this part is that no previous research has used a method for detecting serializability and cycles for unlimited number of transactions accessing the data on routers where the transactions way of accessing the data items on the routers have a gap. In addition to this, linear temporal logic has not been used in this scenario to prove correctness of the network system. This part is very helpful in network administrative protocols where it is critical to maintain correctness of the system. This safety property can be maintained using the presented work where detection of cycles in transactions accessing the data items can be detected by only checking a limited number of cycles rather than checking all possible cycles that can be caused by the network transactions.

The second part of the thesis offers two contributions. Firstly, we specify the basic synchronous network flooding algorithm, for any fixed size of network, in LTL. The specification can be customized to any single network topology or class of topologies. A specification for the termination problem is formulated and used to compare different topologies with regards to earlier termination. We give a worked example of one topology resulting in earlier termination than another, for which we perform a formal verification using the NuSMV model checker. The novelty of the second part comes in using linear temporal logic and the NuSMV model checker to specify and verify the liveness property of the flooding algorithm. The presented work shows a very difficult scenario where the network nodes are memoryless. This makes detecting the termination of network flooding very complicated especially with networks of complex topologies. In the literature, researchers focussed on using testing and simulations to detect flooding termination. In this work, we used a robust technique and a rigorous method to specify and verify the synchronous flooding algorithm and its termination. We also showed that we can use linear temporal logic and the model checker NuSMV to compare synchronous flooding termination between topologies.

Adding to the novelty of the second contribution, in addition to the synchronous form of the network flooding algorithm, we further provide a formal model of bounded asynchronous network flooding by extending the synchronous flooding model to allow a sent message, non-deterministically, to either be received

instantaneously, or enter a transit phase prior to being received. A generalization of ‘rounds’ from synchronous flooding to the asynchronous case is used as a unit of time to provide a measure of time to termination, as the number of rounds taken, for a run of an asynchronous system. The model is encoded into temporal logic and a proof obligation is given for comparing the termination times of asynchronous and synchronous systems. Worked examples are formally verified using the NuSMV model checker. This work offers a constraint-based methodology for the verification of liveness properties of software algorithms distributed across the nodes in a network.

# Acknowledgements

This research was funded by Ajloun National University, Jordan. Many thanks goes to Ajloun National University for providing the funding for this research.

Foremost, I would like to express my deepest appreciation to my supervisors Dr. Amitabh Trehan, Dr. Rafat Alshorman and Dr. Walter Hussak for their guidance, support and kindness through my PhD study journey.

A very special gratitude goes to my parents for their support, encouragement, love and prayers.

My heartfelt thanks go to my wife, children, brothers and sisters and all family members who have been an essential and indispensable source for moral support, selfless support, understanding, with constant love.

Special thanks goes to my supportive friends and to everyone I met and supported me during my PhD journey.

# Contents

<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Overview of the state-of-art . . . . .	4
1.3 Statement of the research problem/knowledge gap . . . . .	4
1.4 Critical evaluation . . . . .	5
1.5 Aim . . . . .	5
1.6 Objectives . . . . .	6
1.7 Contributions . . . . .	6
1.8 Overview of thesis chapters . . . . .	7
<b>2 Research Background</b>	<b>8</b>
2.1 Better Software Engineering . . . . .	11
2.2 Formal Methods . . . . .	12
2.3 Basics of Database and Transactions . . . . .	13
2.3.1 Importance of Concurrency Control . . . . .	15
2.4 Distributed Systems . . . . .	18
2.5 Temporal Logic . . . . .	24
2.5.1 Temporal Properties . . . . .	26
2.5.2 Temporal Operators . . . . .	27
2.5.3 Properties Expressed in Temporal Logic . . . . .	27
2.6 Model Checking . . . . .	29
2.6.1 Stages of Model Checking . . . . .	31
2.6.2 Transactions and Temporal Logic . . . . .	31
2.7 Thesis Structure . . . . .	32
<b>3 Literature review and RM</b>	<b>34</b>
3.1 Introduction . . . . .	34

3.2	Literature Review . . . . .	35
3.3	Motivation . . . . .	38
3.4	Research Methodology . . . . .	39
<b>4</b>	<b>Network routing protocols</b>	<b>40</b>
4.1	Introduction . . . . .	40
4.2	Motivation . . . . .	41
4.3	Methodology . . . . .	41
4.4	Concurrent Transactions and Serializability . . . . .	42
4.4.1	Concurrent Transactions and Histories . . . . .	42
4.4.2	Transactions Accessing Data in the Same Order . . . . .	43
4.5	Description of the Protocol . . . . .	46
4.6	Temporal Logic . . . . .	47
4.6.1	Syntax of LTL . . . . .	48
4.6.2	Semantics of LTL . . . . .	48
4.7	Specification of Routing Protocol in LTL . . . . .	49
4.8	Verification of the Routing Protocol using the NuSMV model checker	55
4.9	Conclusion . . . . .	56
<b>5</b>	<b>Synchronous network flooding</b>	<b>57</b>
5.1	Introduction . . . . .	57
5.2	The Synchronous Flooding Algorithm . . . . .	58
5.3	Linear Temporal Logic . . . . .	60
5.3.1	Syntax of LTL . . . . .	60
5.3.2	Semantics of LTL . . . . .	60
5.4	Specification of the Synchronous Flooding Algorithm . . . . .	61
5.4.1	Edge Propositions . . . . .	62
5.4.2	Send-message Propositions . . . . .	62
5.4.3	Message-received Propositions . . . . .	63
5.4.4	Initial Conditions . . . . .	63
5.4.5	Topological Constraints . . . . .	64
5.4.6	Termination . . . . .	66
5.5	Applications . . . . .	66
5.6	Worked Example . . . . .	70
5.7	Conclusions . . . . .	76
<b>6</b>	<b>Asynchronous network flooding</b>	<b>78</b>
6.1	Introduction . . . . .	78
6.2	An Asynchronous Network Flooding Model . . . . .	79
6.3	Temporal Model . . . . .	83

6.3.1	Propositions . . . . .	83
6.3.2	States . . . . .	83
6.3.3	Run constraints . . . . .	84
6.3.4	Termination . . . . .	85
6.3.5	Rounds . . . . .	85
6.4	Comparing Asynchronous and Synchronous Termination . . . . .	86
6.4.1	Worked Examples . . . . .	87
6.5	Conclusions . . . . .	94
<b>7</b>	<b>Conclusions</b>	<b>97</b>
7.1	Future work . . . . .	98
	<b>Appendix</b>	<b>108</b>
<b>A</b>	<b>Routing protocol encoding into LTL</b>	<b>108</b>
<b>B</b>	<b>Synchronous Flooding Algorithm encoding into LTL</b>	<b>114</b>
B.1	Topology1 terminates before Topology2 with initial node =0 . . . . .	114
B.2	Topology2 terminates before Topology1 with initial node =0 . . . . .	116
B.3	Topology1 terminates before Topology2 regardless of initial node . . . . .	118
<b>C</b>	<b>Asynchronous flooding algorithm encoding into LTL</b>	<b>121</b>
C.1	Asynchronous Can Terminate Before Synchronous . . . . .	121
C.2	Asynchronous Cannot Terminate Before Synchronous . . . . .	123



# List of Figures

1.1	Safety-Critical Systems Controller Diagram. . . . .	2
2.1	A graphic representation of problems resulting from errors in specifications . . . . .	12
2.2	Flooding - line connection. . . . .	20
2.3	Flooding - ring connection. . . . .	22
2.4	Flooding - triangle connection. . . . .	23
2.5	Flooding - 2 ring connection . . . . .	24
2.6	Thesis Chapters . . . . .	33
4.1	Representation of the set of ordered routers. . . . .	47
4.2	Representation of the set of ordered routers with gaps. . . . .	47
4.3	Counterexample on cycle $T_1T_3T_2T_1$ . . . . .	53
4.4	Counterexample on cycle $T_1T_2T_4T_1$ . . . . .	53
4.5	Counterexample on cycle $T_2T_1T_4T_2$ . . . . .	54
4.6	Counterexample on cycle $T_3T_2T_4T_3$ . . . . .	54
4.7	Counterexample on cycle $T_3T_1T_4T_3$ . . . . .	55
4.8	Counterexample on cycle $T_3T_1T_4T_3$ . . . . .	55
5.1	Flooding example 1 . . . . .	59
5.2	Flooding example 2 . . . . .	60
5.3	Flooding on two topologies . . . . .	66
5.4	Flooding rounds in two topologies . . . . .	68
5.5	Two topologies containing five nodes . . . . .	70
5.6	Synchronous flooding model checking block diagram. . . . .	71
5.7	Topology1 terminates before Topology2 with initial node 0. . . . .	74
5.8	Topology2 terminates before Topology1 with initial node 0. . . . .	75
5.9	Topology1 terminates before Topology2 always. . . . .	76
6.1	Example 2.6 . . . . .	81
6.2	Synchronous and Asynchronous flooding model checking block diagram. . . . .	88

6.3	Figure of four nodes . . . . .	91
-----	--------------------------------	----

# List of Tables

2.1	Two transactions executing simultaneously. . . . .	16
2.2	Bus tickets - two transactions executing simultaneously. . . . .	16
2.3	Dirty read problem example . . . . .	17
2.4	Temporal logic operators. . . . .	27

# Chapter 1

## Introduction

This chapter presents a brief background to the research conducted, and an overview of the state-of-art (which will be presented in details in the Literature Review Chapter), statement of the research problem, critical evaluation, aim, objectives, contributions and then an overview to the thesis chapters.

### 1.1 Background

Software is used in different devices and systems to accomplish specific tasks according to the device and environment. Many of the devices that we see around us come with a software which controls it. The software that controls the working of an Information and Communication Device is our interest. examples of such devices include the automated teller machine (ATM) where the user inputs the card key and make a selection of the amount of money required and an output in cash is given to the user. An aeroplane is another example. Inputs can be as location, weather, ...,etc. which will be given to the programs that control the aeroplane in deciding its trajectory. Medical instruments such as a pacemaker which is used to monitor and regulate the heartbeat of a human heart takes inputs as signals of the heart and give outputs by the program as to control the heartbeat. Autonomous cars are also another example of such systems and devices. This type of program which controls these devices is called a controller of the device. The controller listens to inputs and takes decisions and gives outputs as shown in Figure 1.1.

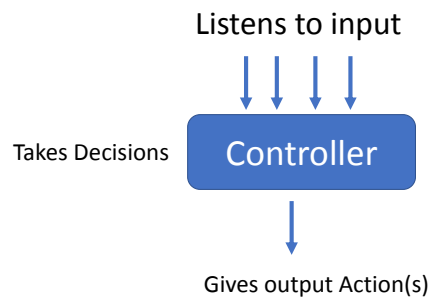


Figure 1.1: Safety-Critical Systems Controller Diagram.

Due to the high costs of errors in safety-critical systems, better software engineering methods need to take place to avoid defects in controllers. While designing controllers for safety-critical systems, it is important to ensure that the controller is reliable and takes correct decisions which cover all possible scenarios. Usually, the controllers come with a set of requirements that they need to satisfy. To check if the controller satisfies its requirements, test cases can be used to find the outcomes, but when the number of components that interact with the controller increases, manual verification becomes increasingly difficult where some errors of the controller can go unnoticed. A verification technology called model checking can solve this problem. The approach to do the verification is by constructing a mathematical model of the device controller and writing its requirements in a formal notation. If the mathematical model of the controller satisfies the requirements written in this formal notation, then this means that the controller satisfies its requirements. This is done automatically which eliminates human errors of manual proofs. Temporal logic proved to be a successful technique in formal methods and can be used to specify the properties of critical systems.

This research focuses on using temporal logic on distributed systems as they are considered reactive systems and have properties to be met. Reliability and safety are two of the main concerns in critical computer-based systems. The jobs running on these systems are not tolerant to errors in output. Concurrency makes these systems very difficult to build and verify, especially in light of the emergence of mobile systems. The transactions within concurrent systems require the use of shared resources. The resources that are accessed and updated by such concurrent transactions should leave the system in a consistent state, as having more than one transaction attempting to access and update similar data items could leave

the system in an inconsistent and unsafe state. If we look at networking systems, we notice that the software used to make sure a certain packet is delivered from its source to its destination, on routers in this instance, is built according to certain routing algorithms. Routing protocols serve to specify the path by which data packets are delivered to the specified destination. Routing protocols are usually tested according to different situations and scenarios, but in these situations which are not tested for there is the possibility that a system could malfunction or become insecure.

This research considers maintaining the safety property, which means that nothing bad will happen, in a distributed system where routers in a network have data items that are shared by unlimited number of transactions. This is considered a new contribution where unlimited number of transactions access data items in with a ‘gap’, which will be defined later in Chapter 4. This contribution provides easier cycle detection after calculating the gap. The contribution here provides correctness for the network routers. Other research mainly focuses on testing/simulations. The method presented here explores all possible system states in a brute-force manner. As concurrency can cause different problems in a system, specifying the system and verifying it should be accomplished in a way which maintains the system safety. Accessing the data items on routers can lead to having the data being modified in an incorrect way which will violate the safety property of the system. The transactions also need to be managed in by certain rules which leads to a safe system state. A network protocol is presented in Chapter 4 to help maintaining the safety property in distributed systems. The research also focuses on the liveness property, something good will happen, in message passing distributed systems. The second and third contributions work with memory-less network flooding where nodes on the network don’t have memory. The well-know synchronous and asynchronous flooding algorithm is specified and verified. The termination of the flooding algorithm is specified and verified using linear temporal logic (LTL) and the NuSMV model checker. In the second contribution, we specify the basic synchronous network flooding, for any fixed size of network, in LTL. The specification can be customized to any single network topology or class of topologies. A specification for the termination problem is formulated and used to compare different topologies with regards to earlier termination. A worked example is given of one topology resulting in earlier termination than another, for which we perform a formal verification using the NuSMV model checker. In the third contribution, we provide a formal model of bounded asynchronous network flooding by extending the synchronous flooding model to allow a sent message, non-deterministically, to either be received instantaneously, or enter a transit phase prior to being received. A generalization of ‘rounds’ from synchronous flooding to the asynchronous case is

used as a unit of time to provide a measure of time to termination, as the number of rounds taken, for a run of an asynchronous system. The model is encoded into temporal logic to compare the termination times of asynchronous and synchronous systems. Worked examples are formally verified using the NuSMV model checker. This work offers a constraint-based methodology for the verification of liveness properties of software algorithms distributed across the nodes in a network.

## 1.2 Overview of the state-of-art

State-of-the-art, formal methods, are used in system specifications, especially in critical systems where the cost of failure could notably go beyond the cost of system development. Formal methods are mathematical entities that can be used to model the system in question, and are used to model system properties in a thorough manner. Formal methods are mathematical-based techniques used for both the specification and verification of software systems. The use of formal methods in software engineering ensures a certain robustness and reliability to the final product. Temporal logic has been used in specifying and verifying different properties of reactive systems. The ability of using temporal logic and model checking to rigorously reason about the specifications properties in reactive systems is the advantage which led to use them in Human Computer Interaction (HCI) field in the computer technology, Air Traffic Control (ATC), autonomous vehicles, trains signalling systems and many other safety-critical systems.

In distributed systems, testing and simulation is the main technique used. As this covers only specific scenarios, faults can happen in the systems. In this research we use formal methods to prove correctness. Mathematical proofs can be done by an expert person in mathematics which for businesses is not always considered. On the other hand, manual proof is liable to human error. In this research we use linear temporal logic and the model checker NuSMV to carry automatic proof which eliminates errors. Temporal logic is used to specify the safety and liveness properties in network algorithms. The method used in this research explores all possible system states in a brute-force manner leading to building rigorous systems.

## 1.3 Statement of the research problem/knowledge gap

Temporal logics have been used in the specification and verification of safety and liveness properties of concurrent transactions in databases systems. This research

investigates the application of linear temporal logic to distributed systems which include shared resources accessed by different transactions. It also investigates the application of linear temporal logic to other networks with message passing and no memory. It look to specify and verify the safety and correctness of concurrency with respect to collections of different transactions accessing data in different ways. It also looks to specify liveness property of memory-less flooding algorithm. The research is a combination of theoretical modelling and practical specification and verification using a temporal logic model checker NuSMV.

## 1.4 Critical evaluation

Linear temporal logic proved to be a good technique in specifying different network algorithms properties. It showed that it can check for safety property as for the first contribution. The livenss property was specified in the second and third contribution using temporal logic. Termination of the synchronous and asynchronous flooding algorithm is specified in this research in linear temporal logic. This shows that linear temporal logic is a powerful method and can be used to specify network algorithms and its properties. In the case that a property was not met, the model checker gives a counterexample showing the states that cause the error. The states represent a trace which can be helpful in defining the error. Linear temporal logic is chosen in this research over other techniques as first-order temporal logic because there are practical and theoretical obstacles to formal verification in such logics. LTL is used also due to the nature of the problem of flooding where all messages that arrive ‘at the same time’ are aggregated into the same round, this has ruled out standard process calculi approaches such as CSP [1] and CCS [2], which only allow two processes at a time to synchronize sending and receiving of messages.

Even with the specifications presented in this research, and the use of one of the most powerful model checkers available, NuSMV, proofs will only be possible in practice for fairly small sizes of network. A strong mathematical background can help in solving and proving some problems where data is accessed in different ways other than what is presented in this work. Some problems faced during this research will need more time and effort to tackle in addition to higher mathematical skills.

## 1.5 Aim

Applying linear temporal logic to specify and verify safety and livness properties in distributed systems.



## 1.6 Objectives

The objectives of this research are as follows:

- Investigate the usage of temporal logic and select appropriate one(s) for the research in addition to selecting a model checker and learning using it.
- Review the literature of the field.
- Find distributed systems problems that can be specified and verified using temporal logic.
- Investigate how data is accessed on routers and if can be modelled.
- Apply theoretical work to the model when matching.
- Use a model checker to verify the properties over the model.
- Provide conclusions and possible future work.

## 1.7 Contributions

This work provides three contributions. The first contribution specifies a model of the execution of an unbounded number of concurrent transactions over time in propositional Linear Temporal Logic (LTL) in order to prove serializability. It uses recent work which places bounds on the number of transactions needed when data items are accessed in order, but do not have to be accessed contiguously, i.e., there may be ‘gaps’ in the data items being accessed by individual transactions. The aim of this contribution is to specify the concurrent modification of data held on routers in a network as a transactional model. The correctness of the routing protocol and ensuring safety and reliability then corresponds to the serializability of the transactions. Verification of the software of administrative routing protocols is expected to be one of the main applications of this work.

The second contribution works on memory-less flooding. We specify the basic synchronous network flooding algorithm, for any fixed size of network, in LTL. The specification can be customized to any single network topology or class of topologies. A specification for the termination problem is formulated and used to compare different topologies with regards to earlier termination. A worked example is given of one topology resulting in earlier termination than another, for which we perform a formal verification using the NuSMV model checker.

The third contribution provides a formal model of bounded asynchronous network flooding by extending the synchronous flooding model to allow a sent message, non-deterministically, to either be received instantaneously, or enter a transit

phase prior to being received. A generalization of ‘rounds’ from synchronous flooding to the asynchronous case is used as a unit of time to provide a measure of time to termination, as the number of rounds taken, for a run of an asynchronous system. The model is encoded into temporal logic. It also compares the termination times of asynchronous and synchronous systems. Worked examples are formally verified using the NuSMV model checker. This work offers a constraint-based methodology for the verification of liveness properties of software algorithms distributed across the nodes in a network.

## 1.8 Overview of thesis chapters

The second chapter will provide a research background. The third chapter provides literature review, motivation, and research methodology. The first, second, and third contributions are presented in Chapter 4, Chapter 5, and Chapter 6 respectively. Chapter 6.5 provides general contributions and future work. Appendices are provided at the end of the thesis.

# Chapter 2

## Research Background

Software complexity has increased rapidly in recent years due to the increased complexity of the systems using the software. To create software that can accomplish certain requirements requires that such software first needs to be specified in the early stages of development as per these requirements, and at a later stage needs to be verified against those requirements to ensure that they have been properly met. With the increase in demand to provide reliable software, especially in critical systems, software engineers must avoid introducing errors into their software and attempt to verify that the requirements specified by the client are met. The software in reactive systems, where the system consists of different parts that react together and with the environment, requires that certain properties must be satisfied, amongst which are safety and liveness. Network systems consist of software which is designed to achieve certain tasks over the network. Different network algorithms are used in these systems to achieve particular tasks, and ensuring that these have been specified and verified using a rigorous method avoids the introduction of errors into these systems. Formal methods, and specifically Temporal Logics, have been used by different systems to specify and verify software so as to achieve the required properties in these systems due to their power in these regards. In this thesis, temporal logic will be used to specify and verify network algorithms with respect to safety and liveness. The safety property will be examined in the first part of the thesis, namely in Chapter 4. The second part of this thesis will examine the liveness property, as discussed in Chapter 5 and Chapter 6. In Chapter 4, we will investigate the use of Linear Temporal Logic LTL to specify and verify the concurrent modification of data on the routers in a network as a transactional model in order to prove serializability. The correctness of the routing protocol, and ensuring safety and reliability, can then be said to correspond to the serializability of the associated transactions. In Chapter 5, we will specify a basic synchronous network flooding algorithm, for any fixed size of network, in Linear Temporal Logic. A specification of the termination problem

is formulated and is used to compare different topologies in terms of earlier termination. Chapter 6 provides a formal model of bounded asynchronous network flooding by extending the model of synchronous flooding to allow a sent message to either be received instantaneously, or enter a transit phase prior to being received, in a non-deterministic manner. The model is encoded into temporal logic and a proof obligation is given in terms of comparing the termination times for asynchronous and synchronous systems. This chapter will give an introduction to the different concepts and techniques that will be used in the thesis in addition to a number of examples of system failures.

Different business, and indeed the public sector, have certain, unique standards for the products they deliver which are ultimately to limit the potential for physical or economic damage. Smartphones users, for instance, would become upset at any faults or failures in their devices because of wrong or unexpected results. However, such system failures clearly do not cause physical harm to the users; rather, they have the potential to cause economic damage. This kind of failure can negatively impact the companies using these systems. For instance, in September 2016 Samsung recalled 2.5 million Note 7 phones due to a manufacturing issue with their batteries that resulted in overheating [3], in some instances resulting in severe burns being inflicted on a large number of customers. The BBC reported that this recall was believed to have cost Samsung \$5.3 bn, who ultimately recalled this smartphone to ensure the safety of its customers. Samsung offered either a replacement or a refund [3].

In stock market trading and e-business, users access data remotely via different forms of transactions in what can be described as a highly mobile environment. For instance, If the stocks trader receives inconsistent data, he/she may accordingly make a poor decision and will be negatively affected in a financial, though not physical, sense. For example, if a transaction has to read more than one data element that is being broadcast by the server to the clients, the client needs to receive consistent and correct data. One might consider a trader exchanging dollars in one country whilst another makes the same exchange in another country through the same exchange company; if the data is not fresh on both sides, they can receive different amounts of money. This example shows the seriousness of having a correctly specified and verified scheduling algorithm. In database management, the system responsible for managing data concurrency and consistency is called the scheduler [4]. Its responsibility is to schedule different concurrent transactions containing reads and writes of a set of data items. Choosing an appropriate and accurate schedule is thus essential to ongoing database integrity.

Critical computer systems are very strict and their failure is not acceptable; the negative impact that might result from a software or operational defect should

be prevented at all costs. The famous chip manufacturer, Intel, lost about 475 million US dollars because of a design fault in its Pentium II processor, which was discovered in 1994 [5]. Patient information can be collected during the normal life of the patient without requiring that they be hospitalized using mobile technologies. Every year we see new health technologies intended to improve patient health and/or lifestyle. Healthcare is becoming increasingly reliant on technology to perform its various operations. When the issue is one of health, such operations become even more critical. At least six cancer patients died due to radiation overexposure caused by the software controlling a Therac-25 radiation therapy machine in 1985-1987 [6]. This bug was a direct result of concurrent programming errors. As a result of software malfunctions, passengers have died in aircraft, car, and train accidents. A fatal Airbus crash in May 2015, where an Airbus A400M crashed in Spain, was because of engine control software issues [7]. In August 2007, Skype experienced a critical disruption due to massive restarts of users [8] as a result of an error in the associated software. This problem was ultimately found to have occurred due to an unexpected number of users trying to concurrently access the systems. The software ultimately had to be updated on users' systems and then reconnected to the service, and because of restarts of those users, the system had a huge number of transactions representing connections to the service by their users [8].

With the increase of the number of the interactive systems, the potential for defects increases exponentially [5]. Specifying and verifying the correctness of the software in such environments should be undertaken to ensure proper fault avoidance. In September 2017, a check-in system failure created chaos at airports across the world [9]. The biggest airports worldwide, including London Heathrow, Charles de Gaulle in Paris, Changi in Singapore and Washington DC's Reagan Airport, amongst others, were affected because of this failure. The check-in software responsible was provided by a company called Amadeus, who specialize in travel technology, confirmed that this failure was a result of a network issue. Passengers affected by this problem had their flights either delayed or cancelled. Airport personnel were unable to provide the details required by these passengers until the system started working again; in such situations, airline companies are required to compensate travellers, which will clearly cost them large amounts of money [9]. The network software in these and other communication systems handle very large amounts of data transmission and large numbers of transactions, and so network algorithms which have not been formally specified and verified could cause unexpected issues at some unknown point in the future.

This research focusses on the specification and verification of network algorithms using temporal logic. The problems that occurred as listed above, and

indeed that have occurred in many other systems, indicate the need to use robust methods of software engineering. This thesis focusses on the software part of communication, in particular on the networking algorithms used in networking systems. The following section will discuss the different areas of software engineering and the importance of the software requirements and the verification part of the software. Section 2.2 introduces the formal methods used to address these purposes and the importance of their use in software engineering. Section 2.3 introduces the definitions used in this thesis with regards to database transactions in addition to describing some of their more important properties. This section represents an important prelude to Chapter 4, which discusses data modification on network routers. Section 2.4 gives an introduction to distributed systems and distributed algorithms, which are discussed further in Chapter 5 and Chapter 6. In Section 2.5, temporal logic is introduced in addition to temporal properties and temporal operators; temporal logic is used in this thesis to specify and verify network problems. After this, Section 2.6 discusses model checking and its importance.

## 2.1 Better Software Engineering

Software engineering starts from the early stages of the software specification, and typically ends with the maintenance of the software. This is of particular importance when considering financial constraints. Tools, theories and methods are used in software engineering to successfully create software with the required standards. The most elementary steps of software engineering consist of software specification, development, validation and evolution [10]. In a business sense, companies will attempt to produce reliable and trustworthy software for their clients while keeping minimizing the revenue required to achieve this goal.

Some of the most important steps in software engineering are the specification and verification of the software. Selecting higher standard specification and verification methods for reactive and concurrent systems implies higher level of dependability. Software specification represents that start of the software engineering process's activities. The specification starts when software engineers meet with their customers and gain a comprehensive description of the software to be developed and its constraints. Poor or incorrect specifications or misunderstandings will lead to a system that does not meet the customer's needs, which in the extreme can lead to financial loss for both the software production company and the user alike [11]. This demonstrates the serious need to form specifications that correctly describe the required system. Later in the system development process, the system will need to be checked to meet the requirements of the customer.

This process is called verification. Verification is tested against the current specifications for the software; in critical systems, this is a crucial step in determining and demonstrating their correctness. Using formal methods and model checking to both specify and verify software is considered to provide reliable software that meets the associated requirements.

Figure 2.1 shows a famous cartoon drawing of creating a swing and reflecting the idea during software development. It shows how incorrect or misunderstood specifications can lead to unintended results.

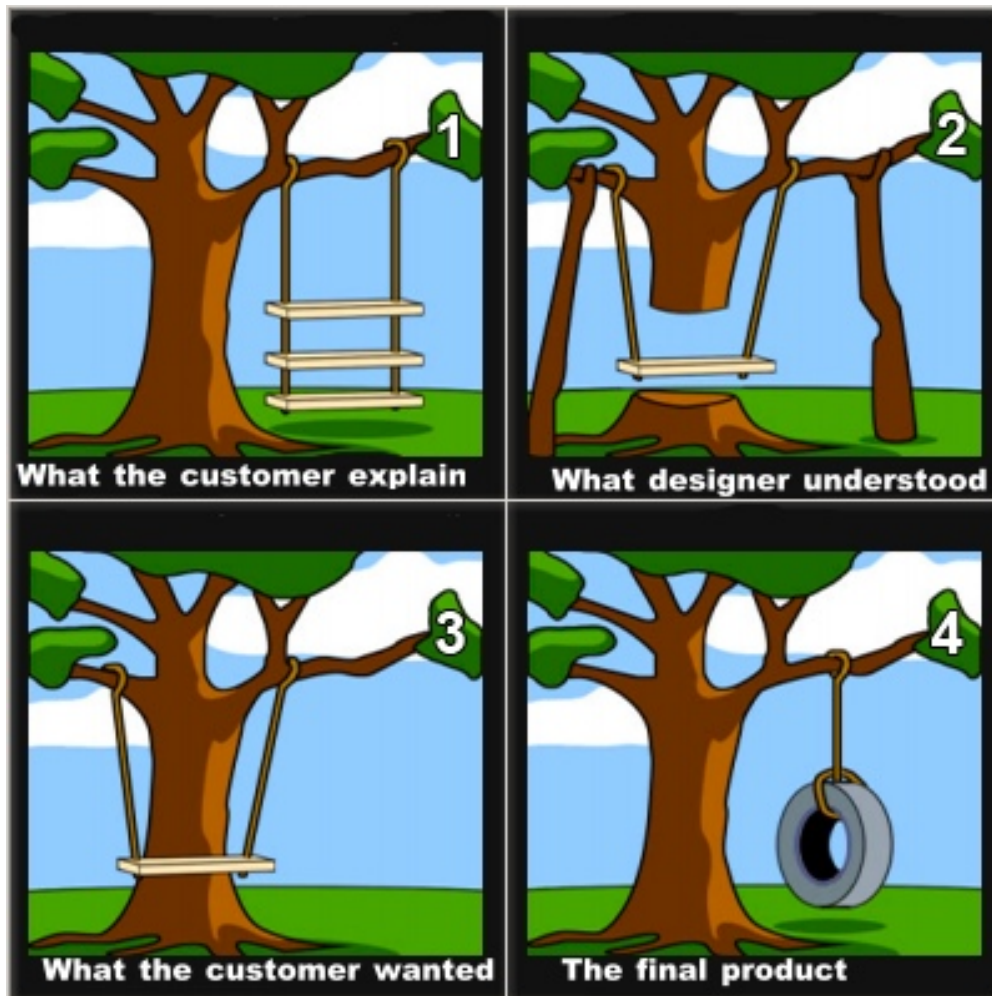


Figure 2.1: A graphic representation of problems resulting from errors in specifications

## 2.2 Formal Methods

State-of-the-art, formal methods, are used in system specifications, especially in critical systems where the cost of failure could notably go beyond the cost of system development. Formal methods are mathematical entities that can be used to model the system in question, and are used to model system properties in a

thorough manner. Formal methods are mathematical-based techniques used for both the specification and verification of software systems. The use of formal methods in software engineering ensures a certain robustness and reliability to the final product.

The quality of the results that can be obtained by applying formal methods in verifying critical systems has led to their extensive use by software engineers. The National Aeronautics and Space Administration (NASA) reported that formal methods should be part of every software engineer's and computer scientist's education [5]. To provide an ultra-detailed verification of the system, the appropriate properties should be defined and specified in a precise manner.

In critical systems, the major approaches used to specify and verify the correctness of the concurrency control protocol are those of mathematical proofs. Mathematical proofs need a person with high levels of experience in mathematics, which makes their use difficult in the software development industry [12]. In addition to this need, mathematical proofs could themselves have errors due to human error. Using automated proofs as available model checkers to model systems formally according to their specifications and to prove their correctness saves time and avoids human error. In this manner, the desired properties can be verified using an exhaustive search of all possible states that the system can enter during execution. Model checkers can provide different and interesting features by observing the different states the system goes through. In NuSMV, in case a certain property is not met, a counterexample is given showing how the system *did not* comply with the property. This research will use formal methods, specifically temporal logic and model checkers, to specify and verify network algorithms and their properties.

## 2.3 Basics of Database and Transactions

This section will provide an introduction to the basics of databases and database transactions; it will also introduce concurrency and the need to control concurrency. This section will be helpful as an introduction to Chapter 4 as the topics discussed here provide an excellent guide to concurrency problems when modifying data on network routers.

A database is defined as a collection of related data which is organized so that it can easily be accessed, managed, and updated [4]. This database can vary according to its type and to different organizations. A transaction is a sequence of database operations performed by the execution of a program. Database transactions entail, for instance, banking functions, reservations, and stock market functions, amongst many others. If a person is to book a flight, they will



very likely access a booking page to check for the availability of flights from their departure to their destination airports. Different airlines may operate at different stages of their journey. The customer will thus be accessing a database that combines different airlines with the different airports on their journey, whilst the booking of the flight itself represents the transaction they make on the database. Once the customer books their ticket, their details will be used to reserve seats on the flight(s) on this journey. The first step, that of searching for the flight, represents a read operation from the database; the booking step represents the write operation. The plane will have certain places available for travellers. Once the traveller books a ticket, the availability of seats for other travellers will, obviously, be reduced. A booking system should run without faults and ensure that certain properties are met, for instance, that no two travellers can book the same seat; if all seats are reserved, the system should not allow any further bookings to be made on that flight. Other properties must hold in such systems where different transactions by different users could potentially attempt to access the database simultaneously. We can imagine how such issues have grown exponentially with today's mobile technologies. A person can nowadays book a ticket for a trip using his/her smart-phone in a very straightforward manner.

Concurrency is one of the topics that has been considered for decades and indeed has grown with new mobile technologies. The scheduler is the database system component that handles concurrency control. The scheduler is an essential component of transaction processing systems that deals with users running concurrent transactions. The scheduler produces schedules which represent the interleaved execution of these transactions. The schedule is also sometimes called a history. A transaction can perform the following database operations:

- Access operation (read operation): fetches the data value of a data item  $x$ . This is denoted by  $\text{Read}(x)$ .
- Update operation (write operation): updates the value of a data item  $x$ . This is denoted by  $\text{Write}(x)$ .

The database is a representation of part of the real world [12, 13, 14]. This means that its state is governed and controlled. An example is that a person's weight cannot be negative. Another example is that only a definite number of passengers can be seated in an aircraft according to its number of seats. Such restrictions are referred to as integrity constraints, which create a framework to ensure that data consistency is valid when any user performs an operation that will result in a change to the database. A database is said to be in its consistent state at a certain time if all data element values are valid according to the integrity

constraints. If a transaction is to commit, it should maintain the database in its consistent state. Four properties of transactions (known as ACID properties) should be valid when transactions are executing in the database to preserve its consistency. Furthermore, in an environment where multiple transactions are executing, the database management system (DBMS) must schedule the concurrent execution of the transactions steps. The schedule of these transactions operations must have the property of being serializable [4]. To understand more of the required properties, we should review these properties and what they mean. The first four properties are the ACID properties:

*Atomicity:* All operations of a transaction are required to be complete or else the transaction is aborted. The transaction is treated as a single unit.

*Consistency:* A transaction is aborted if any part of it violates an integrity constraint. It should keep the database in a consistent state to commit.

*Isolation:* In a concurrency environment, simultaneously running transaction behaviours do not affect each other. Access to shared resources must be serialized by the transactions.

*Durability:* Once a transaction is committed, the changes it makes cannot be undone. Effects should also not be lost, even in the instance of a system failure.

*Serializability:* Results of schedules in an environment where transactions are executed concurrently should maintain a consistent system.

After understanding the different properties, we should consider transaction concurrency, and why it is important to control it, further. This is considered in the following section.

### 2.3.1 Importance of Concurrency Control

In an environment with multiple transactions accessing shared data and that are executed in parallel, concurrency control is needed to avoid any undesirable situation that can violate the consistency of the database. In other words, the DBMS needs to preserve some or all ACID properties in such environments [12, 15]. Without concurrency control, transactions running simultaneously over a shared database can create data integrity and consistency issues. The following example explains the first problem:

Suppose that we have two transactions (Transaction1, Transaction2) accessing a data item  $x$  in a database. The item  $x$  has an initial value of 5 ( $x = 5$ ).

Transaction1:  $Read(x); x = x - 3; Write(x);$

Transaction2:  $Read(x); x = x + 7; Write(x);$

The concurrent execution of the two transactions is represented in Table 2.1.

In Table 2.1, the final value of  $x$  is 12. If the transaction were executed in a se-

Table 2.1: Two transactions executing simultaneously.

Step	Transaction 1	Transaction 2
1	$Read(x)$	
2	$x = x - 3$	
3		$Read(x)$
4		$x = x + 7$
5	$Write(x)$	
6		$Write(x)$ ← older value is cancelled

quential order, the final value of  $x$  would be  $x = 9$ . We can conclude from this that the value yielded by the concurrent transactions is incorrect. It is clear that the reason for this incorrect value of  $x$  is that the second transaction, (*Transaction2*), reads the value of  $x$  before the first transaction (*Transaction1*) changes it in the database. As a consequence, the change resulting from the first transaction is lost (overwritten). This problem is known as the lost update problem. To take another example, let us assume that we have a bus with ( $x$ ) seats reserved ( $x = 9$ ). The total limit number of passengers who can book seats is 16. The first transaction cancels 6 seats reservation ( $x = x - 6$ ). The second transaction reserves seven seats ( $x = x + 10$ ). Hence, the final value of reserved seats should be 13 ( $x = 13$ ). The concurrent execution of the two transactions is represented in Table 2.2.

Table 2.2: Bus tickets - two transactions executing simultaneously.

Step	Transaction 1	Transaction 2
1	$Read(x)$	
2	$x = x - 6$	
3		$Read(x)$
4		$x = x + 10$
5	$Write(x)$	
6		$Write(x)$ ← older value is cancelled

If we look at the interleaving operations of both transactions in Table 2.2, the final value is 19 ( $x = 19$ ). We can conclude from this example that this value yielded by the concurrent transactions is wrong and the reservation limit has been reached (or exceeded). It is clear that the reason for this incorrect value of  $x$  is that the second transaction, (*Transaction2*), reads the value of  $x$  before the first transaction (*Transaction1*) has changed  $x$  in the database. As a consequence, the changes resulting from the first transaction are lost (overwritten).

When a transaction finishes its steps successfully it is said that the transaction has committed successfully. A well-known problem when dealing with the concurrent execution of transactions on a shared data item is known as a dirty read (or temporary update), where a transaction reads a data element which has

been updated by another transaction but where the first transaction has not yet committed [13, 16]. The problem occurs when the transaction that modified the data item has not finished and rolls back for a certain failure. To demonstrate this problem, consider the following example:

Suppose that we have two transactions (Transaction1, Transaction2) accessing a data item  $x$  in a database. The item  $x$  has an initial value of 10 ( $x = 10$ ).

Transaction1:  $Read(x); x = x - 3; Write(x);$

Transaction2:  $Read(x); x = x + 7; Write(x);$

The concurrent execution of the two transactions is represented in Table 2.3.

Table 2.3: Dirty read problem example

Step	Transaction 1	Transaction 2
1	$Read(x)$	
2	$x = x - 6$	
3	$Write(x)$	
4		$Read(x)$ ← Dirty read
5		$x = x + 10$
6		$Write(x)$
7	ABORT (Failure)	

We can see from Table 2.3 that Transaction1 encounters a failure before committing and Transaction1 reads the shared data item  $x$  after Transaction1 has modified it. In this case, Transaction1's failure will cause it to abort and to be rolled back (restarted), while Transaction2 does not restart. Transaction2, in this scenario, will have read a value of  $x$  that is now is never considered to have existed, resulting in an inconsistency in the database. If Transaction1 finishes successfully, however, then the database will remain in a consistent state.

Other problems could arise in a concurrent environment such as an unrepeatable read. The unrepeatable read problem is encountered when the first transaction reads several values and the second transaction updates some of these values while the first transaction is being executed [13, 16]. Such problems can occur in an environment where concurrent transactions executions exist, hence violating the ACID properties. The lost update problem causes a violation in the consistency and isolation properties. This is because the state of the database is inconsistent after executing these transactions.

In Chapter 4, we will focus on specifying and verifying the isolation property, which is the responsibility of the concurrency control system. The isolation property guarantees that any transactions that are executed concurrently will result in a system state similar to the state that would result from transactions being executed serially (i.e., without any transaction interleaving).

We have now provided an introduction to the basics of database transactions, which will be the main topic of Chapter 4 which describes data modification on network routers via concurrent transactions. After having provided this short introduction on the need for concurrency control, we will now introduce distributed systems and some of the concepts of the flooding algorithm which will be the topic of Chapter 5 and Chapter 6.

## 2.4 Distributed Systems

Distributed systems are systems of multiple components that interact with each other. There are two types of distributed systems: shared memory distributed systems, where components share the same memory, for example a multi-core processor in a CPU; and message-passing distributed systems, where the individual components interact with each other through messages sent on links and which corresponds to computer networks. In this thesis, we concentrate on message-passing distributed systems.

The components of message-passing systems use the messages passed between them to communicate and process different tasks. Message-passing systems have characteristics which include the concurrency of components, independent failure of components, and the lack of a global clock. The interaction of the different components and the message passing required between them to accomplish a certain job is considered difficult in terms of the lack of a global clock. Processing the job can be achieved in a parallel manner between the different elements of the distributed system. In a distributed system, the different components all have the same goal as the outcome of their work.

Distributed network routing algorithms deal with directing and redirecting messages between different network routers and end points [17]. In this thesis, routers and endpoints are referred to as nodes. The router's job is to send the message to one of its neighbours - to which it has a connection - in order to deliver the message to its destination. [17]. Distributed network algorithms can be classified as synchronous distributed algorithms, where a 'global' clock is assumed to exist in the system and messages are sent during the same clock tick. The other classification is asynchronous distributed algorithms, where messages are not sent at exactly the same time or during the same clock tick. In this thesis, each of these types are considered in Chapter 5 and Chapter 6 on a well-known network algorithm called the flooding algorithm [18]. The phenomenon of flooding forms the basis of many important distributed processes [19].

The flooding algorithm is an algorithm which utilizes every path in the network [20]. As the algorithm specifies, one of the nodes will start to send a message to

all of its neighbours. When any given node receives a message from a neighbour, it forwards the same message to all of its neighbours except the one(s) it received the message from [21, 20, 18, 22]. Synchronous distributed algorithms assume a ‘global clock’ where actions happen during clock ticks or rounds. This means that the network has bounded link delays and a lockstep synchronization with the pulses of the global clock. In the message synchronization property, a message sent from node  $v$  to neighbour  $u$  at pulse  $p$  of  $v$  must be delivered to  $u$  before pulse  $p + 1$  of  $u$  [20].

In this thesis, we investigate ‘memoryless’ flooding where a node does not explicitly remember if it has previously taken part in the process or which nodes it has previously interacted with. This may happen, for example, if the node does not have enough memory to store its past history or if there are multiple flooding operations occurring which it does not want to, or cannot, distinguish. It does, however, know which node(s) sent it the message in the present round and forwards copies of the message to all its neighbours with the exception(s) of the one(s) it received it from. Notice that if in any round a node receives the message from all its neighbours, the node does not need to do anything. If at some point no node forwards the message, we say that the flooding has terminated. It is hard to know if the flooding process will ever terminate, however, especially in complicated topologies with cycles.

To gain a visual understanding of the synchronous flooding algorithm and how it works, and indeed the termination problem, we present some examples below. The first example will show a sequence of nodes connected in a line, as illustrated in Figure 2.2(a)-(d). A node with a message is shown as double circled.

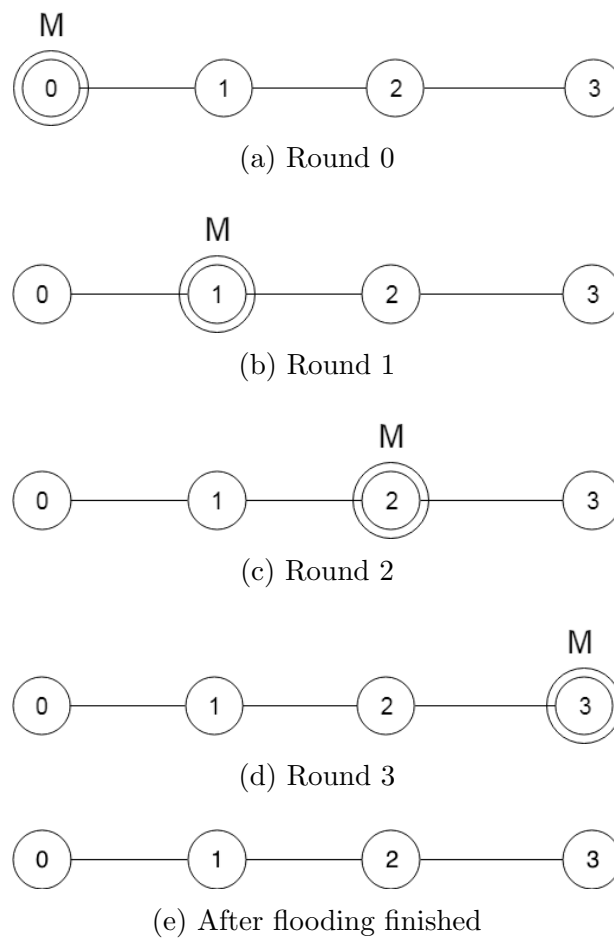


Figure 2.2: Flooding - line connection.

The nodes are connected bidirectionally; this means that each individual line connecting the nodes can carry a message in both directions. In round 0 a message is at node 0, as shown in Figure 2.2(a). In the first round, a message is sent from the initial node 0 to its neighbour, node 1, as shown in Figure 2.2(b). In the second round, each neighbour which received the message will forward this to all of its own neighbours, except for those from which the message was received. As node 1 has two neighbours (node 0 and node 2), it can send the received message from node 0 to node 2 according to the algorithm, as shown in round 2. Eventually, all the nodes in the network will receive a message in a certain round, and where this is achieved in round 3, as shown in Figure 2.2(d). After node 3 has received the message it will not be forwarded to any of 3's neighbours as its only neighbour is node 2, but this is the node from which the message was received. At this stage, the flooding of the message terminates. Flooding on the network of Figure 2.2 shows that the flooding can be terminated. Figure 2.2(e) shows the network after flooding has stopped. If the initial node were any other than the one considered in this example, we can see easily that flooding will again terminate. Let us consider another example of synchronous flooding on a different network, as illustrated in

Figure 2.3. This example shows a ring network of five nodes.

As we can see in 2.3, in round 0 the message is at node 0, as shown in Figure 2.3(a). Node 0 has two neighbours, nodes 1 and 2. In round 1, the message is sent to both of these neighbours. In round 2, the message at node 1 will only be sent to node 3, as it cannot be sent to the other neighbour, node 0, as it received the message from this node originally. Similarly, for node 2, the message will be sent to node 4. As shown in Figure 2.3(c), node 3 and node 4 now have a message. Each of the nodes will send the message in the next round to the neighbour from which it did not receive the message from in the previous round. Node 3 will send its message to node 4 and node 4 will forward its message to node 3 in round 3, as shown in Figure 2.3(d). In the next round (round 4), node 3 will forward the message to node 1 and node 4 will forward the message to node 2, as shown in Figure 2.3(e). In round 5, the message at node 1 will be forwarded to node 0 and the message at node 2 will be forwarded to node 0, as shown in Figure 2.3(f). Since node 0 has received a message from both node 1 and node 2, the flooding will terminate as node 0 does not send to the neighbours it has received from. Figure 2.3(g) shows the network after flooding has terminated.

Another example of synchronous flooding is shown in Figure 2.4.



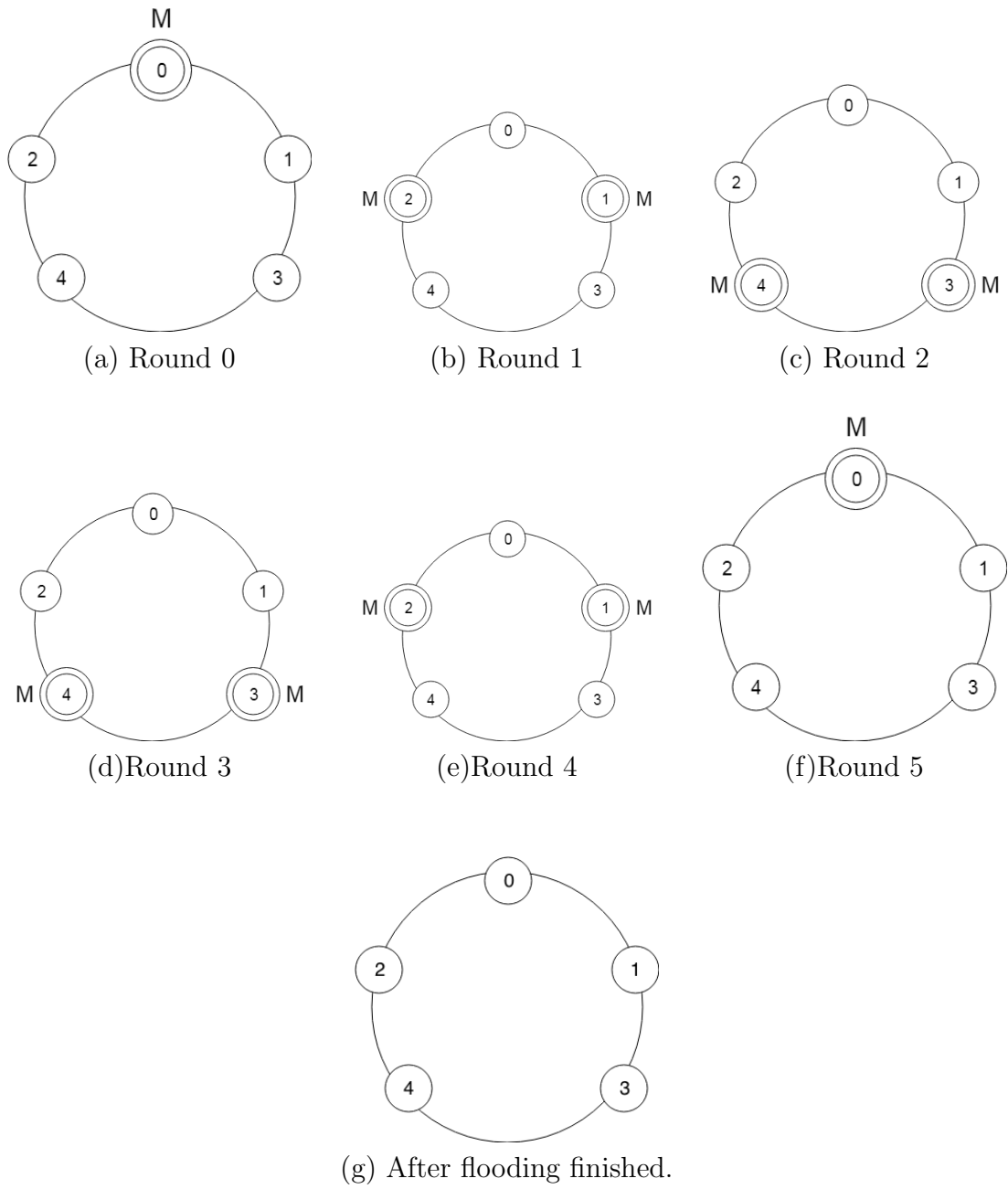


Figure 2.3: Flooding - ring connection.

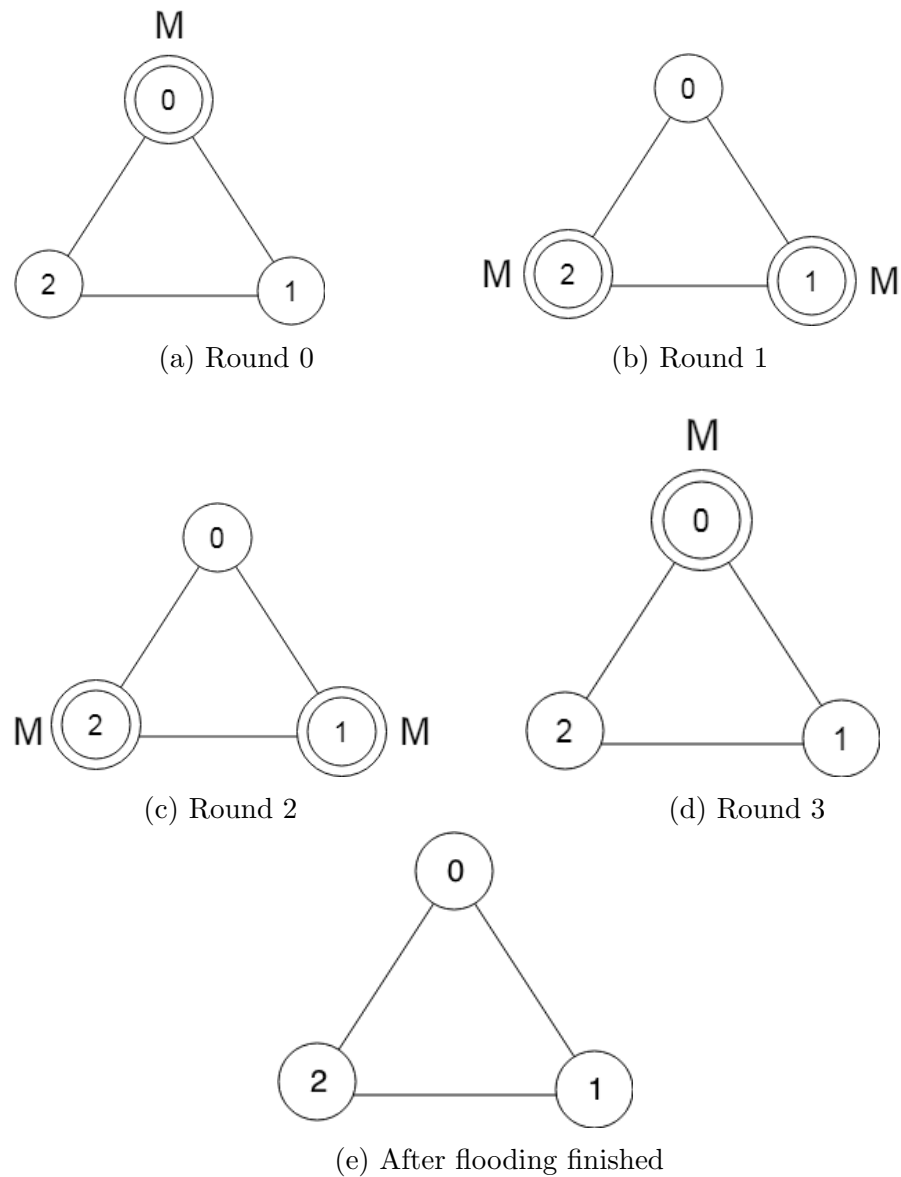


Figure 2.4: Flooding - triangle connection.

We can see, as shown in Figure 2.4(a)-(d), that the message was at node 0 at the start of flooding and was forwarded in each round until the flooding stopped. The message at node 0 was forwarded to node 1 and node 2 in round 1. In round 2, the message was forwarded from node 2 to node 1 and from node 1 to node 2, as shown in 2.4(c). In round 3, the message from node 2 and node 1 was sent to node 0 as shown in Figure 2.4(d). At this stage, the flooding terminates as node 0 will not forward the message to any of its neighbours as it received a message from each of them in the previous round. Figure 2.4(e) shows the network after flooding has terminated. Flooding termination can be more complicated depending on network topology and number of nodes. If we consider Figure 2.5, for example, then it is clear that tracing the flood of messages will be more challenging than in the earlier examples.

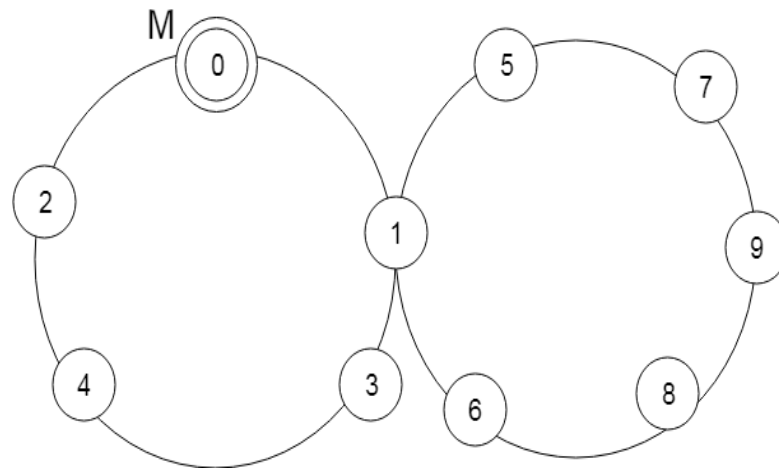


Figure 2.5: Flooding - 2 ring connection

Node 0 is the initial node with the message at round 0. Its neighbours will receive the forwarded message in round 1. Each neighbour receiving the message will forward it to its neighbours in the next round. The flooding will continue to ensure that all nodes in the network receive the message. In this example, flooding will, in fact, terminate, though as implied earlier this is not necessarily the case. The situation is also more challenging when flooding is asynchronous.

If we take the flooding example illustrated in Figure 2.4 and assume in this instance that the flooding is asynchronous, this means that the messages are not sent and delivered at the same time. If the initial message was at node 0 and the message is forwarded to neighbour nodes 1 and 2, in asynchronous flooding delivery to neighbours can occur at different times, which makes predicting flood termination far more difficult. This means that the message can be delivered to one node but not delivered to the other as it is still in transit. The one which receives the message will forward the message to its neighbours according to the flooding algorithm, with the exception of the one it received it from. This scenario can cause loops, hence flooding might not terminate.

## 2.5 Temporal Logic

In mathematics, classical propositional logic formulae are interpreted as truth values, either “true” or “false” [23], and which is absolute. The truth values of the formulae are fixed after the propositional variables have been mapped to the truth values. In reality, we need to consider the changes that happen in our universe. For statements like ‘I am wearing a jacket’ or ‘it is snowing’, it is clear that their truth value can, and likely will, change over time. A person can wear a jacket when it is cold and snowing on one day but will take it off when it is warm or sunny on another. In other words, the valuation of the statements can change

with time. This need to represent changes has led to the introduction of temporal logic.

Temporal logic is used to represent reasoning about a changing world [24] where the formula's truth values may vary over time [12]. The importance of temporal logic is to address time-dependent valuations. Temporal logic is an extension of classical propositional logic. Facts about the past, present, and future states can be expressed in the formulae of temporal logic. Temporal logic has been used extensively in the representation of temporal information because the concept of time is inherent to the logic. Today's computer system requirements are rapidly increasing, especially with the emergence of smart mobile devices including mobile phones and other smart devices that can be used in different places. The varying states of the systems over time require that special tools must be used to specify and verify their states; modelling changes in such time-dependent systems can be achieved using temporal logic. Temporal logic can be used to specify the different properties of a system. A system state which changes over time can be represented using temporal logic formulae. Examples of what temporal logic can represent are encapsulated in the following statements: 'the network is always running', 'a sent message will be delivered in the future', 'if I pay for my order, I will receive it eventually', 'a car system will use the car brakes whenever a car is closer to the car in front'. Temporal logic provides the ability to reason about a time-line. Linear temporal logic (LTL) adopts this type of reasoning. The LTL model of time is like a line, are a so-called path. Computational tree logic (CTL) is a type of temporal logic which includes a branching logic, where time is modelled using a tree-like structure.

Due to the power of temporal logic, specification and verification of concurrent and reactive systems makes significant use of temporal logic [23]. E-commerce and traffic control systems are examples of such systems, where at the same time an error in this type of system is considered fatal. Both software and hardware engineers dealing with critical systems do not rely purely on testing their systems, as these need to be further specified and verified in such a way as to match the desired requirements to minimize the possibility of errors. Demonstrating these requirements in a rigorous manner requires a highly skilled mathematician. An important factor to take into consideration is that a mathematician is still human, and human error is always a possibility. Another problem is that it is not always easy to accomplish the proof if there is a mathematician with high skills. The solution to overcoming this problem is to use automatic model checkers, which will avoid human error. The different properties required of a system can be encoded within the temporal logic of a model checker.

Model checkers that use different types of temporal logic were developed as

they have the ability to verify real-world systems in a short time, which is particularly useful considering these systems have a massive number of states. After the system-required properties are encoded into the temporal logic in the model checker, it then checks all different possible states of the system to verify the properties. In the well-known model checker NuSMV, if a property is not satisfied a counterexample will be given illustrating the states of the system where the property is *not* satisfied. Temporal logic is helpful for specifying concurrent systems by describing event ordering over time. With the exponential growth of technologies and concurrent systems, these systems have become more complicated and their interactions more critical. This means that the specification and verification of their properties is essential [25, 26]. The following subsection discusses these different properties, as followed by Subsection 2.5.2 which introduces the operators used in temporal logic. Subsection 2.5.3 shows how to express system properties in the form of temporal logic.

### 2.5.1 Temporal Properties

Reactive and concurrent systems are usually very complex in structure, making their design and analysis processes very difficult to achieve. A mistake in the design can easily occur in such difficult system processes. Any mistake can lead to unwanted properties, such as deadlock, occurring. Safety, liveness, and fairness are the major properties of concurrent and reactive systems that require specification and verification [26, 27, 28].

These properties can be described as follows:

**Safety:** The safety property requires that the system will not have anything bad happening while it is running. Mutual exclusion, freedom from deadlock and partial correctness are examples of this property. The following statements are examples of safety properties: ‘the reactor temperature will never reach  $150^{\circ}C$ ,’ ‘the car will never start as long as the key is not in the ignition position.’ Again, the property assures that nothing bad will happen.

**Liveness:** The liveness property assures that something good will eventually happen. This property is important to ensure that something ‘good’ must happen in the future. Liveness properties are important where there are any infinite behaviours of the systems. Examples of the liveness property are total correctness, guaranteed accessibility and responsiveness. The following statements are examples of liveness properties: ‘the traffic light will turn green,’ ‘the message will be delivered eventually,’ ‘the program will terminate.’

Fairness: The fairness property ensures that a request must be granted. This property helps to serve services via a fair strategy. The advantage of defining fairness properties is to avoid reaching properties without specifying the fairness properties. Fairness constraints are imposed to avoid forcing the system to perform unrealistic computations. These constraints are unconditional fairness, strong fairness and weak fairness. Unconditional fairness imposes the condition that every process can be executed infinitely often; the strong fairness property states that every process that is enabled infinitely often gets to be executed infinitely often; and the weak fairness property ensures that every process that is continuously enabled from a certain point in time can be executed infinitely often. It is important to understand that fairness is a requirement of demonstrating liveness [5, 29]. The different types of fairness property describe liveness properties.

## 2.5.2 Temporal Operators

There are two types of operators in temporal logic. The first are the ordinary logical operators ( $\wedge, \vee, \neg, \Rightarrow, \iff$ ) which have their usual meanings, whilst the second are temporal operators which are used in temporal logic, such as LTL (Linear-Time Temporal Logic) and CTL (Computational Tree Logic). Temporal logic takes into consideration the necessity and the possibility concepts. If  $\beta$  is a formula, then  $F\beta$  is a temporal logic formula that asserts that  $\beta$  is possibly true, and  $G\beta$  is a temporal logic formula that asserts that  $\beta$  is necessarily true. Table 2.4 summarizes some of the temporal logic operators' meanings:

Table 2.4: Temporal logic operators.

Operator	Meaning
$F\alpha$	$\alpha$ will be true at some time in the future.
$G\alpha$	$\alpha$ will always be true in the future.
$\alpha U \beta$	$\alpha$ will always be true until $\beta$ becomes true.
$X\alpha$	$\alpha$ will be true next.
$E\alpha$	Exists: there exists at least one path starting from the current state where $\alpha$ holds.
$A\alpha$	All: $\alpha$ has to hold on all paths starting from the current state.

## 2.5.3 Properties Expressed in Temporal Logic

This section will express some of the system properties mentioned earlier in subsection 2.5.1 using temporal logic [12].

**Safety properties:**

## 1. Mutual exclusion:

No two processes can access the same resource simultaneously. For example, if two processes  $\alpha$  and  $\beta$  are running asynchronously (only one of them take a step at any given moment) and the order of execution is undetermined. Mutual exclusion is described in temporal logic as follows:

$$G\neg((\alpha = R) \wedge (\beta = R))$$

where  $\alpha = R$  means that the process  $\alpha$  uses the resource  $R$ .

## 2. Freedom from deadlock:

At least one process is allowed to progress at any time. This can be written formally as:

$$G(enabled_1 \vee \dots \vee enabled_k)$$

where  $enabled$  is true if process  $i$  has an action that can be executed (for  $1 \leq i \leq k$ )

## 3. Partial correctness:

After the program starts, if  $\alpha$  is satisfied, then  $\beta$  will be satisfied if the program reaches a successful state  $\gamma$ .

$$\alpha \Rightarrow G(\gamma \Rightarrow \beta)$$

**Liveness Properties:**

## 1. Guaranteed accessibility:

A process that is in a particular state will eventually go to the next state. For example, the computations that execute both process  $\alpha$  and process  $\gamma$  infinitely often will hold:

$$G((\alpha = i) \Rightarrow F(\alpha = (i + 1)) \wedge G((\beta = i) \Rightarrow F(\beta = (i + 1)))$$

where the processes  $\alpha$  and  $\beta$  can be in state 1 (i.e.,  $\alpha = i$ ) then move to the next state, state  $i + 1$ .

## 2. Responsiveness:

A request will eventually be granted upon request. This property can be described as:

$$G(\alpha \Rightarrow F\beta)$$

where  $\alpha$  is a request and  $\beta$  means granted.

3. Total correctness:

After the start of the program, if  $\alpha$  is satisfied, then the program terminates in a state  $\gamma$  where  $\beta$  is satisfied.

$$\alpha \Rightarrow F(\gamma \wedge \beta)$$

**Fairness properties:**

1. Strong fairness:

A process which is enabled infinitely often will be executed infinitely often:

$$\bigwedge_{1 \leq i \leq k} (GF \text{ enabled}_i \Rightarrow GF \text{ executed}_i)$$

This is interpreted as an event that becomes enabled infinitely often (but may become disabled) must be executed infinitely often.

2. Weak fairness:

Any process that is enabled almost everywhere is executed infinitely often, such that:

$$\bigwedge_{1 \leq i \leq k} (FG \text{ enabled}_i \Rightarrow GF \text{ executed}_i)$$

This is interpreted as a constantly enabled event must occur infinitely often.

3. Unconditional fairness:

Every process is executed infinitely often, such that:

$$\bigwedge_{1 \leq i \leq k} (FG \text{ enabled}_i)$$

This is interpreted as being that the process can be executed at any time.

## 2.6 Model Checking

In critical systems, it is a matter of great importance to ensure correctness of both software and hardware as errors or failures have the potential to result in large financial losses and can lead to fatal consequences, especially in safety-critical systems [30, 31]. Verification techniques' formal methods have become of considerable interest when building high assurance systems and in avoiding failures in critical system. The most successful technique used by both the industry and in research is model checking.



Model checking is an automatic verification method for finite state concurrent systems to check if a system model  $M$  or a protocol satisfies its formal specifications as written in logic as temporal logic [30]. The model represents all possible behaviours of the system [5, 12]. The properties of the system are written as formulae. A property formula  $\phi$  is checked by exploring all possible system executions in the state space of the model to demonstrate whether the correctness of the system is satisfied by the model. This is represented as  $M \models \phi$  [32]. An advantage of model checking is that it is fully automatic and does not require particular expertise in mathematics to run or interpret. The automatic tool which achieves this job is called the model checker. Another advantage of model checking is that if a property is not satisfied (i.e., an error is found), a counterexample is given showing the reason for the problem and the state of the system which led to this error [31, 12].

In software engineering, the costs of testing software can range from 30% to 50% of the total cost of the software development [5]. Test generation and test execution can be automated in some areas, but the comparison is usually carried out by human beings. Correctness is determined by making the software travel across a set of execution paths, but ensuring the exhaustive testing of all paths is not possible, which is a big disadvantage in cases where only software testing is carried out. This means that testing can never be complete. On the other hand, in model checking, correctness is checked by an exhaustive exploration of the state space of the model, which makes model checking the rigorous method of choice for use with concurrent and critical systems. State explosion is considered to be the main disadvantage of model checking; when the system has a large number of interacting components or when the system data structure contains a large number of differing values, state explosion can occur because of the huge number of states the system can potentially adopt. The size of the system becomes a problem when it grows exponentially as a result of an increase in the number of state variables. For example, in a system which is composed of  $n$  processes and each process has  $k$  states, the possible number of states by the asynchronous composition of these processes can be defined by  $m^k$ .

For the past 30 years, researchers have tried to solve the state explosion problem so as to be able to provide better model checking approaches. Two main solutions to avoid the state explosion problem are used: the first is to reduce the size of the state space to be searched, and this is generally accomplished using abstraction; the second, which was first introduced in 1987 by Ken McMillan, is to use Binary Decision Diagrams (BDDs) to represent the state space [33]. The latter solution made it possible to verify systems that have more than  $10^{20}$  states [34]. Researchers concentrated on further refinements to BDD-based techniques

subsequent to this accomplishment, and in which the number of states can be more than  $10^{120}$  [35]. The well-known model checker NuSMV is based on these ideas.

This allowed model checking to be successfully used in verifying larger systems than could previously have been attempted, and to successfully detect otherwise highly obscure errors in communication protocols and hardware controllers [31].

### 2.6.1 Stages of Model Checking

The phases of model checking are as follows:

- **Modelling:** In this phase, through the use of the formal description language of the model checker, the design is converted into an acceptable form by the tool, where this form is called the model.
- **Specification:** In this stage, the design properties are written.
- **Verification:** This step includes verifying the specifications against the design to determine whether the specifications are valid or otherwise. In this stage, an exhaustive search of the model state space is carried out using the model checker, which determines if the specification has satisfied or otherwise. If the specification is not satisfied, a counterexample is instead given by the model checker.

### 2.6.2 Transactions and Temporal Logic

It is clear that temporal logic is particularly powerful when dealing with the reasoning adopted in concurrent systems. We can use temporal logic to conduct automatic proofs of this kind of systems to avoid the errors often inherent to manual mathematics proofs. To this end, there are various powerful model checkers available such as NuSMV [36] and SPIN [37], but we need to select which temporal logic is supported by these model checkers. CTL and LTL are supported in these model checkers and can be used in this case.

Serializability is considered to be a safety property [38]. Temporal logic can be used to specify other properties of the histories of an unlimited number of transactions, such as starvation. In *starvation*, some transactions are not served for an indefinite period of time while the system is executing [13]. In [39, 40], starvation is considered a liveness property. The most common problems that face database transaction schedulers are deadlock and starvation [13]. Using temporal logic, deadlock-freedom (which is considered as a safety property) can be achieved if required, especially in transactions that have the potential to iterate an unlimited number of times.

## 2.7 Thesis Structure

In the following chapter, the literature review and research methodology will be presented. In Chapter 4, LTL is used to specify a model of the execution of an unbounded number of concurrent transactions over time in order to demonstrate serializability. This chapter benefits from recent research in specifying the concurrent modification of data on routers in a network as a transactional model.

Chapter 5 specifies the basic synchronous network flooding algorithm, for any fixed size of network, in Linear Temporal Logic. A specification of the termination problem is formulated and used to compare different topologies in terms of earlier termination. A worked example is given for one topology which results in an earlier termination than another, and for which we perform a formal verification using the NuSMV model checker.

In Chapter 6, a formal model of bounded asynchronous network flooding is given by extending the ideas expressed in Chapter 5 with regards to synchronous flooding to allow a sent message to either be received instantaneously, or enter a transit phase prior to being received, in a non-deterministic manner. A generalization of the ‘rounds’ in synchronous flooding is made for the asynchronous case is used as a unit of time in order to provide a measure of time for the termination of a run of an asynchronous system in terms of the number of rounds taken. The model is encoded into temporal logic and a proof obligation is given for comparing the termination times of asynchronous and synchronous systems. We give further related work in the related chapters.

Chapter 7 provides a conclusion to the thesis. Figure 2.6 provides a summary of the remaining chapters in this thesis. The stars next to the chapters indicate that they include the contributions made by this thesis.

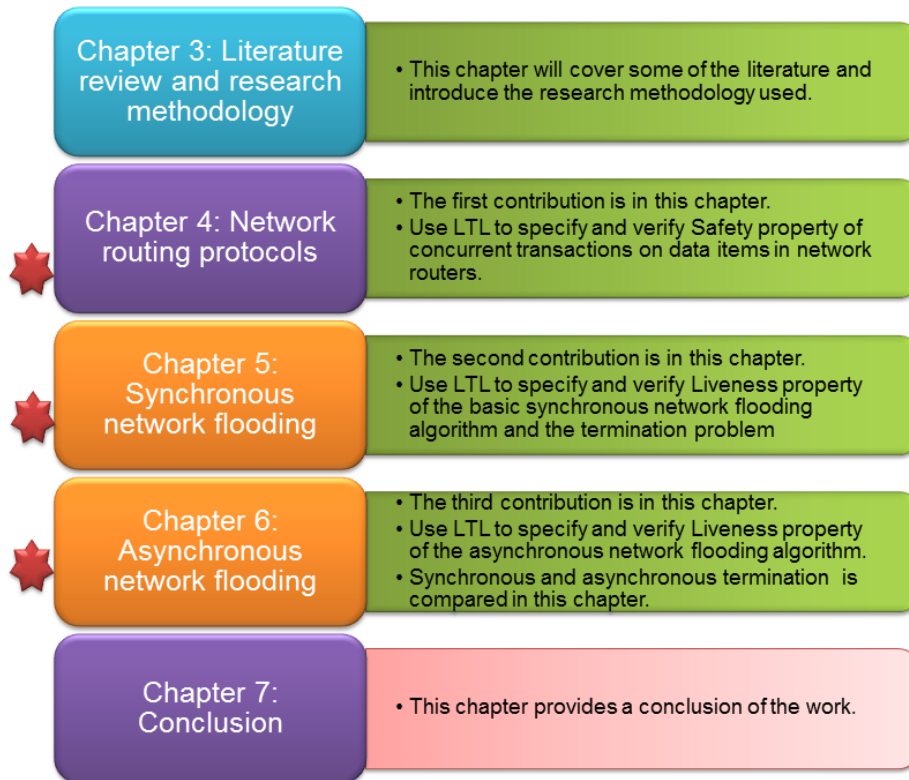


Figure 2.6: Thesis Chapters

# Chapter 3

## Literature Review and Research Methodology

### 3.1 Introduction

With the exponential increase of capabilities of current technology, the need to provide reliable systems has increased in parallel. The safety property (nothing bad ever happens) and liveness property (something good will eventually happen) specifications and verifications for reactive systems have increased to avoid malfunctioning systems and losses of various types. Coordinating the work of the different tasks and components of these systems is important to ensure a consistently running system.

The reliance on database transactions has increased markedly due to the rapid increase in the technology and number of users with access to this technology. Conventional database concurrency methods model finite transaction schedules [12, 14, 41]. The representation of histories of an unlimited number of transactions, as a model, can be achieved using temporal logic. Distributed network systems consist of certain properties, and their algorithms should be specified to maintain these properties. Conventional testing methods cannot physically cover all the potential possibilities that might arise when designing the systems, especially where these systems consist of multiple components that interact with each other in a certain manner. Providing rigorous methods of specifying and verifying these systems to eliminate errors using mathematical proofs is difficult as it requires people with mathematical expertise to maintain such proofs. Even with using mathematical proofs, simple human error means that accomplishing such proofs is, in any case, impossible. An advantage of using temporal logic is that there is no need for special expertise to ensure rigorous verification. Another advantage is the exhaustive checks that can be performed by existing model checkers,

such as NuSMV [36] and Spin [37].

## 3.2 Literature Review

Formal methods, in general, are considered to be powerful when being used to build critical systems in order to make sure that they are robust and secure. Temporal logic in particular has been used in specifying both hardware and software systems. Due to the availability of powerful model checker tools, temporal logic can be used to specify required properties of systems and to subsequently verify them.

Temporal logic has been, is currently is, used in specifying and verifying the various properties of reactive systems. The ability to use temporal logic and model checking to rigorously reason about the specifications properties in reactive systems is the major advantage which led to their use in Human Computer Interaction (HCI) in the computer technology [42]. As user interfaces are considered reactive systems as they interact with their environment, in [43, 44], temporal logic was used to formally specify and verify software user interfaces to minimize the possibility of introducing bugs in software as the possibility increases with the growth of the software user interface, which becomes harder to test for the existence of such bugs. In some fields, such as in Air Traffic Control (ATC), testing can be very expensive and indeed determining the number of test cases that are sufficient to ensure an exhaustive analysis can be very difficult [42]. Using temporal logic and model checking in these types of scenarios can decrease the the number of tests needed, as well as giving the ability to test the various possible system states [42].

Electronic systems used on artificial satellites and aircraft are called avionics which is a term coming from ‘aviation electronics’. These systems control many different operations and functions of the devices which they are used in. The operation of this type of systems has to maintain certain properties at all times. Temporal logic has been used in avionics software which is concerned with safety and reliability properties in avionics [45]. Temporal logic is also used in robotics so that properties and the system model are correct. In health-care, temporal logic has been used due to the high risk to any error that can be caused by a bug in a device or a program [45]. Temporal logic has been used in traffic control [46]. To avoid congestion in traffic control, some statements need to be implemented. An example of these statements is a statement like ‘always avoid traffic’ can be implemented using temporal logic. Testing is not enough in these kinds of systems as testing only shows presence of bugs not their absence.

Concurrency is a topic that has been considered for a number of decades, and indeed has seen increased interest in accordance with the rise in new mobile technologies. The data accessed by any transaction should qualify and meet certain

conditions upon completion of the transaction (when the transaction commits). Different algorithms are used in this field to accomplish this task. In [12], a representation of how multi-step transactions can access data items infinitely many times is given.

In [47], the concurrency control between mobile transactions and update transactions was studied and a protocol proposed to ensure the serializability of schedules. Its main goal was to ensure data consistency and maximize data currency for mobile transactions. An extensive simulation was undertaken and its performance compared with the results of other methods. In [48], a timestamp-based concurrency control protocol was used to maintain the data consistency of broadcast transactions. This research also made an attempt to reduce the abort rate to minimize concurrency control. The results of associated simulation were compared to another protocol to demonstrate the efficiency of protocol proposed therein. Another protocol was developed in [49] for broadcast transactions that allowed hopeless transactions to be discarded. The main goal was to decrease transaction restart rates and increase system throughput, as compared with other protocols, through the simulation results. All of these protocols relied on the use simulations for their protocols, which could potentially contain undiscovered errors. By contrast, we will use model checking, which explores all possible system states in a brute-force manner.

Research into modelling infinite histories was initially reported in [50], which covered transactions that repeated infinitely often. In [12], modelling infinite histories of multistep transactions was studied for mobile transactions. This research used linear temporal logic (LTL) in specifying the properties of these transactions. Partial-ordered temporal logic (POTL) was used to specify concurrent database transactions in [51], whilst the specification of this type of transaction was achieved using quantified-propositional temporal logic (QPTL) in [52]; in [53], LTL was used. A monadic fragment of first-order temporal logic was used to specify the concurrent transactions in [54]. All of these logics are of exponential space complexity and, with the exception of LTL, are at worst undecidable. An advantage of LTL is the ability of available model checkers to use it; a disadvantage of these logics, again with the exception of LTL, is that it is impractical to demonstrate even basic serializability [12].

Distributed network routing algorithms deal with directing and redirecting messages between the different network routers and end points [17]. The router's job is to send the message to one of its neighbours with which it has a connection in order to deliver the message to its destination [17]. A fundamental algorithm which can also be used for routing is the flooding algorithm [18]. Flooding forms the basis of many important distributed processes, for example, the construction

of BFS trees which are used in the work on distributed Leader Election [19]. The flooding algorithm utilizes every available path in the network [20]. In flooding, a message is sent from one node to all of its neighbours. Those neighbours which receives a message will forward the message to all of its neighbours except the one(s) from which it was originally received [21, 20, 18, 22].

Dealing with timing in processes events is very important to determine the sequence of events timing. When dealing with events within a process, the local clock can be used to determine the timing of different events. The difficulty comes when dealing with distributed processes, where time synchronization is very difficult. Processes in distributed systems communicate with each other using messages. the event of sending a message from one process to another leads to an event of a message being delivered at the recipient process. An event of a message being received happens only after an event of sending a message originally happened at the sender. If we have two processes,  $a$  and  $b$ , and process  $a$  sends a message to process  $b$ , we can say that if process  $b$  receives a message from process  $a$ , then process  $a$  must have sent the message to process  $b$  before process  $b$  received it.

To solve the problem of time synchronization in distributed systems, in 1978, a computing scientist, Leslie Lamport, introduced what is called logical clocks to synchronize processes in distributed systems [55]. These clocks can be used to record causality which means that some events in a distributed system must always occur before other events, which means that the event that happened before caused the event that happened after to occur. The ‘happens-before’ logical relationship among pairs of events is denoted by  $\rightarrow$ . We can represent a message  $m$  sent from process  $a$  to process  $b$  as:  $send(m) \rightarrow receive(m)$  According to Lamport’s solution, a timestamp is associated with every event that happens in every process across the entire distributed system. This is achieved by having a local clock (or counter) in each process and associating a timestamp with every event in that process. If we have two processes  $P_i$  and  $P_j$  with local clocks  $C_i$  and  $C_j$ , respectively, and  $P_i$  sends a message to  $P_j$  where  $a$  is the send event and  $b$  is the received event, then  $C_i(a) < C_j(b)$ . This is achieved by setting  $C_j(b)$  to the following:  $C_j(b) = \max(C_i(a) + 1, C_j(b))$ .

Expanders are a very important class of graphs (having the property of being simultaneously sparse and well connected) that have applications in various areas of computer science and mathematics; for instance, in the design and analysis of communication networks, cryptography, error-correcting codes, pseudorandomness, complexity, coding theory, metric embeddings, etc. (for details, see this well-known survey [56]). For example, in the context of distributed computer networks they have been used for building censorship-resistant networks [57, 58], fault toler-



ant networks [59], efficient (Byzantine) agreement and leader election algorithms [60, 61, 62, 63], analysing information spreading, etc. [64]. Thus, even the efficient construction (in static or dynamic fault-tolerant settings) of expander networks is an important line of research [65, 66, 67, 68, 69].

Distributed systems can be specified as a combination of the specifications used for constituent components by using well-studied process calculi approaches such as CSP [1], CCS [2], the  $\pi$ -calculus [70], the Ambient Calculus [71] and I/O automata [72]. These methods are useful when components have significant internal actions/states that affect external actions but which need to be abstracted away to demonstrate the properties of their external behaviour. In this thesis, our interest is in the network flooding algorithm where individual components, in this case physical nodes in a network, have a minimal number of internal states. The global properties that must be demonstrated derive their complexity from the topology of the associated network graph. It may be difficult to achieve a desirable global topology as some form of composition of components, and may necessitate further proofs to show that the topology has indeed been achieved. Here, we choose a more direct logic-based approach, specifying the overall system - algorithm and network topology - as a set of temporal logic constraints in order to demonstrate the required properties. This has the added benefit that a different network topology can, if required, be easily specified by changing a single constraint, rather than many components, in order to achieve the same effect.

### 3.3 Motivation

One of the most difficult challenges for software engineering is to manage the complexity of the algorithms and software found in concurrent systems. Network systems have come to prominence in many aspects of modern life, and therefore software engineering techniques for treating concurrency in such systems has gained in importance. Considerable effort was expended in previous research in the attempt to increase performance in concurrent system environments and network algorithms. The specification of network algorithms and their safety and liveness properties through the use of formal methods is the ultimate aim of the research presented in this thesis. Temporal logic has proved to be a successful technique when used in formal methods, and which has practical application due to the availability of powerful model checking tools such as the NuSMV model checker. We will investigate the specification and verification of network algorithms using temporal logic and model checking. In the first part of this thesis, we will demonstrate the safety property with regards to the data consistency or serializability of a model describing the execution of an unbounded number of concurrent transac-

tions over time which use temporal logic. The particular interest here is that these concurrent transactions could represent software schedulers for an unknown number of transactions being executed across a network. The second part focusses on specifying and verifying the liveness properties of networked flooding algorithms.

### 3.4 Research Methodology

The objective of this research is the specification and verification of network algorithms using linear temporal logic. The main reason for using linear temporal logic is due to the fact that this method can be extended in order to verify the histories of an unlimited number of transactions. In addition to this, another added benefit is that a different network topology can be easily specified by changing a single constraint, rather than many components, in order to achieve the same effect. With the developments of model checkers, NuSMV is powerful tool when dealing with a large number of states and in verifying real-world systems [73]. In Linear Temporal Logic (LTL), time is modelled by representing it as a path where the future is determined. It is used in specifying general reactive and concurrent systems [74, 75].

In this research, we will model the network algorithm as finite state transition systems, where the specification is expressed in LTL. Afterwards, automatically the state space of the state transition system is going to be explored to verify if the protocol or algorithm satisfies the desired specifications. The model checking is guaranteed to terminate due to the finite nature of the model. Considering the power of the model checker and the important element that if any of the specifications do not hold that a counterexample is given [36]. In this thesis, the specifications and verification of network algorithms will be modelled as a finite state machine in NuSMV input language. The protocol transactions and the network algorithm processes are created based on the behaviour of the protocol or the algorithm over time. The model will be identified by a set of desired properties to ensure that the NuSMV model matches these properties, which will be expressed in LTL. Finally, if the specification of the desired property satisfies all system behaviours, the model checker will produce *TRUE*; otherwise, a counterexample will be given by the NuSMV model checker, representing an error source.

# Chapter 4

## Network Routing Protocols

### 4.1 Introduction

Current internet routing protocols differ according to the algorithms used. This chapter focusses on systems of routers where the the routers are accessed by concurrent transactions. The transactions access and update the routers' data; it is vital that this data remains consistent. Because of the high concurrency of such systems, they are considered system critical, where their failure can lead to considerable losses. Since different transactions attempt to access the routers at the same time, it becomes difficult to deal with them due to the importance of maintaining the integrity of the data being accessed. A protocol is presented in this chapter that aims to achieve the consistency condition for concurrent transactions, namely serializability. This protocol is checked for cycles in the conflict graph in terms of the concurrent transactions accessing the data on routers [76]. Due to the availability of powerful model checking tools, temporal logic is used to specify and verify this protocol. Since routers are naturally ordered in some given manner, the transactions access them in an ordered manner. A transaction can access a set of routers, and indeed can skip routers in the set. The routers so skipped represent 'gaps' in the set of routers accessed. By knowing the size of such gaps in the different transactions, an upper bound can be placed on the number of transactions that need to be considered and a serializability condition can be formulated which can then be verified by the model checker.

In the next section, we will present a review of relevant previous work. In section 4.2, we will discuss concurrent transactions and serializability, in addition to discussing 'gap theory', which underlies our work. In section 4.3, we will describe the methodology involved in using temporal logic. Section 4.4 presents the transactional model used in this chapter and gives a detailed description of how gap theory is used. Section 4.5 describes the protocol. In section 4.6, we

formally define LTL and in Section 4.7 we specify the protocol in LTL. The LTL specification is coded into NuSMV in Section 4.8 and verifications are made using the NuSMV model checker, and the subsequent results reported. The last section gives a number of concluding remarks regarding the sections mentioned above.

## 4.2 Motivation

This chapter focusses on specifying and verifying a protocol for an unlimited number of multi-step transactions accessing a finite set of routers with different properties using temporal logic and a model checker. Serializability in concurrent systems is considered a particularly challenging topic in the field of computer systems. Due to the strict properties of critical computer-based systems and, in recent years, the increased number of mobile transactions, our work introduces a routing protocol that can efficiently detect any breach of serializability should the routers be accessed by multi-step transactions with gaps. By calculating the sizes of the gaps in the different transactions, a cycle can be easily detected. This is because there will be no need to check all the different-sized cycles possible. Our work benefits from previous research which defined gaps and introduced a theorem to calculate gap size [77]. This chapter applies this theorem, as introduced in [77], to routing systems. We model the protocol in temporal logic (LTL) and use the NuSMV model checker to prove or indeed that the models satisfy the serializability property.

## 4.3 Methodology

The objective of this chapter is to specify the correctness, in terms of the serializability, of concurrent transactions executing on routers' data, using specifications written in LTL. The purpose of using temporal logic, such as LTL, is that the method can verify an unlimited number of transaction schedules. The significance of temporal logic in computer science is clear, especially in the specification and verification of critical computer-based systems. The availability of model checkers, such as NuSMV, that can be used to model temporal logic properties, and their capability to dealing with a large number of states and verifying real-world systems, allow us to verify the correctness of the proposed protocol. To gain a fully automated verification, the NuSMV model checker will be used to verify the protocol properties specified in temporal logic. The protocol presented will be modelled in terms of finite state transition systems whose specifications are expressed in LTL. The next step will be to explore the state space of the state

transition system, where it is possible to automatically check if the protocol satisfies its specifications, or otherwise. Using LTL, it is possible to express the properties that must be fulfilled by the system. The model checker will be used in the final stage, where it will return a “true” result if the specifications of the required properties have indeed been satisfied; a counterexample is given by the model checker to indicate any potential source of error.

## 4.4 Concurrent Transactions and Serializability

A transaction is a sequence of operations performed on one or more databases which is representative of a single real-world transition. This section introduces some of the basics of concurrent data transactions and their histories. In particular, we will be concerned here with an unlimited number of transactions creating an unlimited number of histories. However, the number of live transactions at any given point in time will clearly be limited to some finite integer,  $n$ .

### 4.4.1 Concurrent Transactions and Histories

**Definition 4.4.1** *A multi-step transaction [10]  $T_i$  is formed of a sequence of read and write steps on data items from a totally ordered set of data items  $D_i = \{x_1, x_2, \dots, x_m\}$ , where every read step  $r_i(x)$  comes before its corresponding write step  $w_i(x)$ , so that*

$$T_i = r_i(x_1)w_i(x_1)\dots r_i(x_m)w_i(x_m).$$

*We assume an infinite set of such transactions,  $T = \{T_i : i \in \mathbb{N}_1\}$ , where  $\mathbb{N}_1$  is the set of positive integers over all time.*

*A schedule, or history, is the sequence of all the steps of the transactions in  $T$*

$$h = \dots, s_j, \dots, s_{j'}, \dots,$$

*such that each step  $s$  of a transaction  $T_i$  occurs at most once in  $h$ , and any step  $s'$  that comes before  $s$  in  $T_i$  comes before  $s$  in  $h$ . The order of the steps in  $h$  is denoted  $<_h$ , so that if a step  $s_j$  occurs before a step (of a possibly different transaction)  $s_{j'}$  in  $h$ , we have the condition that  $s_j <_h s_{j'}$ .*

*A history is serial if all the operations of transaction appear together in  $h$ , i.e., for all steps  $s_j, s_{j'}, s_{j''}$  in  $h$  and  $i \in \mathbb{N}_1$ , if  $s_j <_h s_{j''} <_h s_{j'}$  and  $s_j$  and  $s_{j'}$  are steps of  $T_i$ , then  $s_{j''}$  is also a step of  $T_i$ .*

*Given a history  $h$ , the conflict graph  $G(h)$  of  $h$  is a directed graph whose nodes are equal to the set of transactions  $T = \{T_i : i \in \mathbb{N}_1\}$ , and, for all  $i, j \in \mathbb{N}_1$ , there is an edge from node  $T_i$  to node  $T_j$  iff one of the following conditions is satisfied:*

- (i) a write step in  $T_i$  occurs in  $h$  before a read step of  $T_j$  to the same data item;
- (ii) a read step in  $T_i$  occurs in  $h$  before a write step of  $T_j$  to the same data item;
- (iii) a write step in  $T_i$  occurs in  $h$  before a write step of  $T_j$  to the same data item.

Two different transactions are said to be conflicting if they require access to a shared data item and at least one of their operations is a write operation (on the shared data item). This conflict can leave the database in an inconsistent state when transactions are running concurrently. To avoid a resulting inconsistent database, some form of scheduling the concurrent transactions is needed. A history  $h$  is said to be serializable if it is equivalent to some serial schedule of the transactions (see [14]). In this case, the results of executing the transactions yields the same result as if they were executed in a serial order. Serializable histories allow greater concurrency than serial histories, leading to higher throughput. It is well known that in the case of finitely many transactions a history  $h$  is serializable iff its conflict graph has no cycles. In [12], Theorem 2.4 shows that in the case of infinitely many transactions accessing a finite number of data items, the serializability of their history  $h$  also corresponds to acyclicity in the conflict graph  $G(h)$ .

#### 4.4.2 Transactions Accessing Data in the Same Order

Serializability of histories can be proved for finitely many transactions in polynomial time [12, 13, 14, 15] by showing that there are no cycles of the transactions in the conflict graphs. The problem of searching for cycles in conflict graphs when there is an unlimited number of transactions is that of deciding which finite number of transactions might form a cycle, as there are infinitely many possible choices. This problem was addressed in [12], where it was found that if there is a fixed global total order  $<_D$  on the finite set of all data items  $\{x_1, \dots, x_m\}$ ,

$$x_1 <_D \dots <_D x_m,$$

accessed by the infinitely many transactions, and transactions are only allowed to access contiguous data items that respect this order, then the conflict graph  $G(h)$  of an infinite history  $h$  has a cycle if, and only if, it has a cycle of length 2 (Theorem 3.7 of [12]). So, if transactions are all of the form:

$$T_i = r_i(x_{i_1})w_i(x_{i_1})r_i(x_{i_1+1})w_i(x_{i_1+1}) \dots r_i(x_{i_{m_i}-1})w_i(x_{i_{m_i}-1})r_i(x_{i_{m_i}})w_i(x_{i_{m_i}})$$

a history has a cycle iff there is a cycle of length 2, i.e., we only need to consider the two transactions and two data items that cause the conflict. This somewhat

restrictive condition on data access was relaxed in the case of infinitely many transactions generated by a fixed finite number  $n$  of transactions iterating infinitely many times in [77], so as to require that the order of access by transactions respects the global order but that there could be ‘gaps’ in the sequence of data items accessed. So, for example, if we have the globally ordered data items

$$x_1 <_D x_2 <_D x_3 <_D x_4,$$

, a transaction could, for example, access  $x_2$  then  $x_4$  (here there is a gap as  $x_3$  is not accessed - but this is allowed). However, a transaction could not access, for example,  $x_3$  and then  $x_2$  as the global order of access  $<_D$  would not be respected. The bound in [77] on the length of cycles in the conflict graphs when there are gaps in the succession of data items being accessed by transactions is given below. We will use the notation in Definition 2.5 of [77], where if a set of data items  $D' \subseteq D$  is denoted by  $\{x_a, \dots, x_b\}$ , which will mean that  $x_a <_D \dots <_D x_b$ . Firstly, we define the ‘gap’ in the data items accessed by a transaction.

**Definition 4.4.2** *Assume that the transaction  $T_i$  accesses a set of data items  $D_i \subseteq D$  such that*

$$D_i = \{x_a, \dots, x_c\}$$

where  $x_a <_D \dots <_D x_c$ . Then, the gap  $G^i$  of the set  $D_i$  can be calculated as follows:

$$G^i = (c - a + 1) - (|D_i|) \quad (4.1)$$

where  $(c - a + 1)$  is the number of elements in the sequence  $x_a \dots x_c$ , and  $|D_i|$  is the cardinality of the set  $D_i$  [77].

Secondly, the maximum gap  $G$  of  $k$  ( $\leq n$ ) transactions is defined as follows:

**Definition 4.4.3** *Let  $\{T_i : 1 \leq i \leq n\}$  be a set of transactions that iterates an unlimited number of times to constitute infinitely many transactions  $T = \{T_i : i \in \mathbb{N}\}$ , and let each  $T_i \in T$  access a set of data items  $D_i$ . At any given point in time there exist  $k$  transactions, where  $1 \leq k \leq n$ , such that*

$$\bigcup_{i=1}^k D_i = \{x_a, x_{a+1}, \dots, x_u\}$$

where  $x_a <_D \dots <_D x_{a+1} \dots <_D x_u$ . The maximum gap  $G$  can then be calculated as follows:

$$G = \begin{cases} 0, & \forall i, 1 \leq i \leq k, G^i = 0 \\ u - a - 1, & \exists i, 1 \leq i \leq k, G^i \neq 0. \end{cases} \quad (4.2)$$

The main result is Theorem 3.4 in [77], which states the following:

**Theorem 4.4.4** *Assume  $D$  to be an irreflexively totally ordered set of data items such that*

$$D = \{x_1, x_2, \dots, x_{n-1}, x_n\}$$

*which is accessed by a set of transactions  $T$  generated by  $n$  transactions iterating an unlimited number of times, as in Definition 4.4.3, and that access the set  $D$  as per Definition 4.4.3. Assume we have a maximum gap  $G = n - 2$  in the set  $D$  denoted by  $G_{n-2}$  and that there is a cycle in the corresponding conflict graph  $G(h)$ . There then exists a cycle of length  $n$ , denoted by  $C_n$ , in the corresponding conflict graph  $G(h)$ .*

The following example will explain how we use these definitions and theorems. Assume that we have five transactions,  $T_1, \dots, T_5$ , accessing the set of data items  $D = \{x_1, x_2, \dots, x_8\}$ , as in Definition 2.5 of [77], as follows:

$$\begin{aligned} D_1 &= \{x_2, x_3\} & D_2 &= \{x_3, x_4\}, \\ D_3 &= \{x_3, x_6\} & D_4 &= \{x_3, x_5\}, \\ D_5 &= \{x_4, x_5\}. \end{aligned}$$

First we calculate the *gap*  $G^i$  using equation 4.1 in Definition 4.4.2 as follows:

$$\begin{aligned} G^1 &= 3 - 2 - 2 + 1 = 0, & G^2 &= 4 - 3 - 2 + 1 = 0, \\ G^3 &= 6 - 3 - 2 + 1 = 2, & G^4 &= 5 - 3 - 2 + 1 = 1, \\ G^5 &= 5 - 4 - 2 + 1 = 0. \end{aligned}$$

After finding the gaps, we find the maximum gap. The maximum gap is  $G = 3$  ( $G^3$ ). Hence, by Theorem 4.4.4, the maximum cycle will be of length 5 ( $C_5$ ). Building a precedence graph for the history  $h$  of all transactions  $T_1, \dots, T_5$  will determine if we have a cycle of length 5 ( $C_5$ ). This method will be used in specifying and verifying the serializability of a routing protocol, where we will be able to detect a cycle in an efficient manner. Theorem 4.4.4 (Theorem 3.4 in [77]) can be used in many applications, as discussed in [77]. One of the more important applications discussed in [77] is that of booking a flight e-ticket through different agencies. It is clear that destinations are naturally ordered. Booking a ticket from any place to another will present different options, one of which is where a person can book a direct ticket, meaning that there will be no transit and thus creating a gap. Another option is when a ticket is booked that has multi-stop destinations. In the second situation, there could also be gaps. In the case where the ticket contains all the stops on the path from departure to destination, there is no gap. In



the following section, we will bear this example in mind when considering routing protocols.

## 4.5 Description of the Protocol

An important class of routing protocols is that of administrative protocols, which we intend to model herein. In our protocol, we will assume that we have different routers connecting different systems, where the packets are passed through some, or all, of them from the sending node to the receiving node. The router's job is to create a path where the packet will travel from source to destination. The path consists of the different routers it passes through until it reaches its destination. The routers are naturally ordered in some manner, where they are presented as a set  $D$ . Each router has its own information table. We denote each router by  $r_i \in D$ . A packet can travel through two or more routers according to the path it is set to travel through. A packet path can be set where the packet can reach the destination by going from the first and closest router (source) to the router which is closest to the destination (destination) without going through any other routers. Another scenario is when the path through which a packet must travel to reach its destination consists of more than two routers. The router table needs to be updated, through which data is kept consistent. The different concurrent transactions associated with sending data across a network require this concurrency where a transaction can check (read) the router or update (write on) the router. A scheduler needs to be used to avoid conflicting transactions from stopping the network or corrupting data while sending it across the network. The scheduler in the proposed protocol needs to maintain serializability. The main two steps in the transactions that we will be using in our protocol are the read step and the write step, where accessing and updating the associated data occurs accordingly.

It is clear that the routers are naturally ordered in some manner. Therefore, creating a path from router  $A$  to router  $F$  can include many choices in terms of the other routers included in the path. In our model, we assume that there can be either a direct or multi-stop path between two given routers. To represent this scenario, we assume that we have the set routers  $D$ , which here we will call *destinations*, where  $|D| = k$  are ordered as per Figure 4.1. The set  $D$  contains all destinations starting from location (router)  $A$  to end at location  $F$ . The next location from location  $i$  is  $x_i$ , such that  $x_i \in D$ , as illustrated in Figure 4.1. The first option in creating a path is that the path runs directly from  $A$  to  $F$  without passing through any other destinations. We call this a direct path without stop. Transaction  $T_1$ , which represents this case, accesses the set  $D_1 = \{A, F\}$ . The

gap for the set  $D_1$  is  $G^1 = k - 2$ . This path is illustrated as edge 1 in Figure 4.2. The second choice is to have a path from  $A$  to  $B$  then from  $B$  to  $F$ , which is represented by edges 2 and 3 in Figure 4.2. A third choice might be to select a path from  $A$  to  $B$ , then from  $B$  to  $C$ , and finally from  $C$  to  $F$ . A fourth path which can be selected might be by going from  $A$  to  $B$  to  $C$  to  $D$  to  $E$ , and then finally from  $E$  to  $F$ ; this path contains all the stops from the initial point  $A$  to the final point  $F$ . Here we represent the read step of the transaction as accessing the destination (router), whilst the write step represents the modification on this router. The set  $D$  represents the ordered routers as defined in Definition 2.5 of [77]. The number of ignored destinations from the start to the end destination is represented by the gap, as given in Definition 4.4.2. The maximum gap illustrates the number of destinations that not accessed in the path where there is any available path from  $A$  to  $F$ .

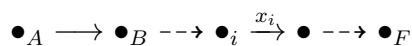


Figure 4.1: Representation of the set of ordered routers.

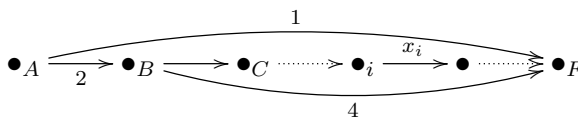


Figure 4.2: Representation of the set of ordered routers with gaps.

## 4.6 Temporal Logic

Temporal logic is used to provide reasoning about a changing world [24], where the formula truth values may vary over time [12]. Facts about past, present, and future states can be expressed in the formulae of temporal logic. The use of temporal logic for formal specification and verification of computer systems was introduced by Amir Pnueli [78]. Temporal logic has been broadly used in the representation of temporal information because the concept of time is built into it. Verification of concurrent and reactive systems makes extensive use of temporal logic [23]. E-commerce and traffic control systems are examples of such systems where, at the same time, an error in this type of system is considered fatal. Model checkers for different types of temporal logic have been developed with the

ability to verify real-world systems in only a short period of time given that these systems generally contain a very large number of states. Temporal logic is helpful in the specification of concurrent systems by describing the event ordering over time. With the exponential growth of technologies and concurrent systems, these systems have become more complicated and an understanding of their interactions more important. This means that the specification and verification of some of their properties are essential [25, 26]. In a paper called “What good is temporal logic?”, which is considered to be of particular significance in the field, Leslie Lamport emphasized that the main function of temporal logic lies in modelling concurrent systems [79]. We will use Linear Temporal Logic (LTL) in the specification and verification of our routing protocol.

### 4.6.1 Syntax of LTL

The alphabet of LTL consists of a set of propositional symbols  $p_i$ ,  $i = 0, 1, 2, \dots$ , which in our use will include special read/write step propositional symbols  $r_i(x_j), w_i(x_j)$ , with  $i \geq 1$  and  $j \geq 1$ , booleans  $\neg, \vee, \wedge, \top, \perp$ , and temporal operators X, F, O, G, U. Formulae in LTL are those generated by:

$$\phi ::= p_i | r_i(x_j) | w_i(x_j) | \neg\phi | \phi_1 \vee \phi_2 | \phi_1 \wedge \phi_2 | X\phi | F\phi | O\phi | G\phi | \phi_1 U \phi_2$$

The symbols  $\top$  and  $\perp$  will also be used to denote true and false values, respectively. The symbols  $\Rightarrow$  and  $\Leftrightarrow$  have their usual logical meanings.

### 4.6.2 Semantics of LTL

Linear Temporal Logic is interpreted over a sequence of states  $s_0, \dots, s_a, \dots$  ( $a \in \mathbb{N}$ ). An interpretation of LTL,  $I(s_a)$ , at a given state  $s_a$  assigns truth values  $p_i^{I(s_a)}$ ,  $r_i(x_j)^{I(s_a)}$  and  $w_i(x_j)^{I(s_a)} (\in \{\perp, \top\})$  to the propositional symbols  $p_i$ ,  $r_i(x_j)$  and  $w_i(x_j)$ , respectively. A (Kripke) structure  $M$ , as defined in [5], is a sequence of interpretations  $I(s_0), \dots, I(s_a), \dots$  for the sequence of states. The semantics of an LTL formula  $\phi$  is given by a truth relationship  $M, s_a \models \phi$ , which means that  $\phi$  holds at state  $s_a$  in the structure  $M$ . The relation  $\models$  is defined inductively as follows:

$$\begin{aligned}
M, s_a \models p_i &\text{ iff } p_i^{I(s_a)} = \top \\
M, s_a \models r_i(x_j) &\text{ iff } r_i(x_j)^{I(s_a)} = \top \\
M, s_a \models w_i(x_j) &\text{ iff } w_i(x_j)^{I(s_a)} = \top \\
M, s_a \models \neg\phi &\text{ iff } M, s_a \not\models \phi \\
M, s_a \models \phi_1 \vee \phi_2 &\text{ iff } M, s_a \models \phi_1 \text{ or } M, s_a \models \phi_2 \\
M, s_a \models \phi_1 \wedge \phi_2 &\text{ iff } M, s_a \models \phi_1 \text{ and } M, s_a \models \phi_2 \\
M, s_a \models \mathbf{X}\phi &\text{ iff } M, s_{a+1} \models \phi \\
M, s_a \models \mathbf{F}\phi &\text{ iff there exists } k \geq a \text{ such that } M, s_k \models \phi \\
M, s_a \models \mathbf{O}\phi &\text{ iff there exists } k \leq a \text{ such that } M, s_k \models \phi \\
M, s_a \models \mathbf{G}\phi &\text{ iff, for all } k \geq a, \quad M, s_k \models \phi \\
M, s_a \models \phi_1 \mathbf{U}\phi_2 &\text{ iff there exists } c \geq a \text{ such that } M, s_c \models \phi_2 \text{ and, for all } a \leq b < c, M, s_b \models \phi_1
\end{aligned}$$

## 4.7 Specification of Routing Protocol in LTL

In this section, we will specify the routing protocol properties. Assume we have four transactions  $T_1, T_2, T_3$  and  $T_4$  accessing four ordered sets of routers  $D_1, D_2, D_3$  and  $D_4$ , respectively. These transactions are iterated infinitely often to generate an infinite history. The data items in the sets represent the different routers. Here, we represent our protocol using four different transactions accessing the ordered routers in such a way as to produce a gap in the accesses. The router item sets are as follows:

$$D_1 = \{x_3, x_4\}, D_2 = \{x_3, x_4\}, D_3 = \{x_3, x_5\}, D_4 = \{x_4, x_5\}.$$

The transactions are as follows:

$$T_1 : \{begin_1, r_1(x_3), w_1(x_3), r_1(x_4), w_1(x_4), end_1\};$$

$$T_2 : \{begin_2, r_2(x_3), w_2(x_3), r_2(x_4), w_2(x_4), end_2\};$$

$$T_3 : \{begin_3, r_3(x_3), w_3(x_3), r_3(x_5), w_3(x_5), end_3\};$$

$$T_4 : \{begin_4, r_4(x_4), w_4(x_4), r_4(x_5), w_4(x_5), end_4\};$$

Here, we have a gap  $G^1$ , where one data element ( $x_4$ ) is skipped in transaction  $T_3$ , and can create cycles of sizes 2, 3 and 4. Accordingly, we can tell that we will have a cycle with a length of 3 and we will only be checking for this cycle. The transactions arrive at the scheduler  $S$  in the order  $T_1, T_2, T_3$  and then  $T_4$ . The

semantics of the formula  $\varphi$  are given by a truth relation  $M, s_i \models \varphi$ , where  $M$  is a (Kripke) structure for *LTL* which satisfies the basic properties of the histories given in (P1)-(P5) below. Given a state  $s_i$ , these properties will yield a matching sequence of steps of reads and writes which becomes true in  $s_i, s_{i+1}, \dots$ . These will be used in conjunction with a LTL specification of the following routing protocol history,  $h$ :

$$h = r_1(x_3)w_1(x_3)r_3(x_3)w_3(x_3)r_2(x_3)w_2(x_3)r_2(x_4) \\ w_2(x_4)r_1(x_4)w_1(x_4)r_4(x_4)w_4(x_4)r_4(x_5)w_4(x_5)r_3(x_5)w_3(x_5))$$

The property (P6) will be the LTL formulae that specify the existence of a cycle. We also add to the beginning and ending of the transactions the propositions  $begin_i$  and  $end_i$ , respectively, indicating when a transaction begins and ends.

Using temporal logic operators, we encode the properties (P1) - (P6) of the protocol in LTL. We present an encoded LTL code for every property for all the transactions below. We only present an example of each property, where this example is to unfold the LTL formula in NuSMV. The remaining unfoldings are described in the following section. The properties of the protocol are as follows:

**(P1) No two reads without a write in-between**

Any transaction which has completed a read step to one data item cannot read another data item without having it write to the first one prior to the second read. If  $x <_D y$ , which means that  $x$  precedes  $y$  in the data item domain,  $D$ ,  $r_i(y)$  cannot be executed before  $w_i(x)$  [73]. This property is to maintain the structure of multi-step transactions as per Definition 4.4.1. We can encode this property into the LTL formula as follows:

$$\sigma_1 = \bigwedge_{i \geq 1} \bigwedge_{x, y \in D_i, x <_D y} G[(r_i(x) \Rightarrow F(w_i(x) \wedge F(r_i(y)))]$$

Taking the case of  $T_1$ , this means that  $T_1$  cannot read  $x_3$  and  $x_4$ , where  $x_3 <_D x_4$ , without having first written to  $x_3$ . This is encoded into LTL in NuSMV as:

LTLSPEC G (T1=r1x3 -> (F (T1=w1x3 & F (T1=r1x4))))

We can also write it in another way as:

LTLSPEC G (((T1=r1x4) & 0(T1=r1x3)) -> 0(T1=w1x3))

**(P2) A write step happens if an item was read**

A transaction  $T_i$  can only write to  $x$  if it has read  $x$  beforehand [73].

$$\sigma_2 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[(w_i(x) \Rightarrow O(r_i(x)))]$$

Taking the case of  $T_1$ , this means that if  $T_1$  accomplished a write step on  $x_3$ , it must have previously read  $x_3$  to ensure that we have read and write steps to each data item  $x$ . This is encoded into LTL in NuSMV as follows:

LTLSPEC G ((T1=w1x3) -> 0(T1=r1x3))

**(P3) A step remains true until the next operation of the same transaction itself becomes true**

No changes will be made to a read/write step until the next operation in  $T_i$  becomes true, i.e., if  $r_i(x)/w_i(x)$  is true, it is unchanged until the next step, where  $x <_D y$ , becomes true [73].

$$\sigma_3 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow \neg(r_i(x))] \wedge \bigwedge_{i \geq 1} \bigwedge_{x, y \in D_i, x <_D y} G[r_i(y) \Rightarrow \neg(w_i(x))]$$

If  $T_1$  reads  $x_3$ , then  $r_1(x_3)$  stays true until  $T_1$  has written to  $x_3$ , at which point  $r_1(x_3)$  becomes false and  $w_1(x_3)$  becomes true. After that, if  $T_1$  needs to read another data item, say  $x_4$ , then  $w_1(x_3)$  becomes false and  $r_1(x_4)$  becomes true. This property is encoded into LTL in NuSMV as follows:

LTLSPEC G(((T1=w1x3) -> !(T1=r1x3)) & G((T1=r1x4) -> !(T1=w1x3)))

**(P4) Each successive state includes only one occurrence of a step**

This is adopted so as not to have two different steps that are false in a given state, and after that the same steps are true in a subsequent state [73].

$$\begin{aligned} \sigma_4 = \bigwedge_{\substack{i, i' \geq 1 \\ 1 \leq j, j' \leq m \\ i \neq i', j \neq j'}} G[ & \neg((\neg(r_i(x_j) \wedge \neg r_{i'}(x_{j'})) \wedge X(r_i(x_j) \wedge r_{i'}(x_{j'}))) \\ & \wedge \neg((\neg r_i(x_j) \wedge \neg w_{i'}(x_{j'})) \wedge X(r_i(x_i) \wedge w_{i'}(x_{j'}))) \\ & \wedge \neg((\neg w_i(x_j) \wedge \neg w_{i'}(x_{j'})) \wedge X(w_i(x_j) \wedge w_{i'}(x_{j'})))] \end{aligned}$$

This property emphasizes the fact that if, say, transaction  $T_1$  reads item  $x_3$ , then it cannot simultaneously write and read in the next step; that is, only one successful step can happen in each state. This is written in LTL for  $T_1$  in NuSMV as follows:

LTLSPEC G ((T1=begin1) -> X!((T1=r1x3)&(T1=w1x3)))

LTLSPEC G ((T1=r1x3) -> X!((T1=w1x3)&(T1=r1x4)))

LTLSPEC G ((T1=r1x4) -> X!((T1=w1x4)&(T1=end1)))

**(P5) Any given transaction can read and write only once to a data item** [73]

For all  $x \in D_i$ , a transaction  $T_i$  can only read data item  $x$  once and only write to data item  $x$  once.

$$\sigma_5 = \left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G \neg [r_i(x) \wedge F(\neg r_i(x) \wedge F r_i(x)) U \text{end}_i] \right) \\ \wedge \left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G \neg [w_i(x) \wedge F(\neg w_i(x) \wedge F w_i(x)) U \text{end}_i] \right)$$

This means that a transaction can only access the data item once for both the read and write steps in a given history. If transaction  $T_1$  writes on data item  $x_3$  having previously read  $x_3$ , it is not allowed to read  $x_3$  again until transaction  $T_1$  ends. This is encoded into LTL as follows:

$$\text{LTLSPEC } G ((T1=w1x3 \ \& \ 0(T1=r1x3)) \rightarrow (F!(T1=r1x3))) U (T1=end1)$$

### (P6) There is a cycle of length 3

The conflict graph of the routing protocol is serializable if there is no cycle in the conflict graph  $G$  of a history  $h$  that is generated by the protocol. Since we found a maximum gap  $G = 1$  ( $G^1$ ), this means that we only need to check for a cycle of length 3 ( $C_3$ ). If we find this cycle, this means that the history is not serializable. To achieve this, we use the following LTL formula:

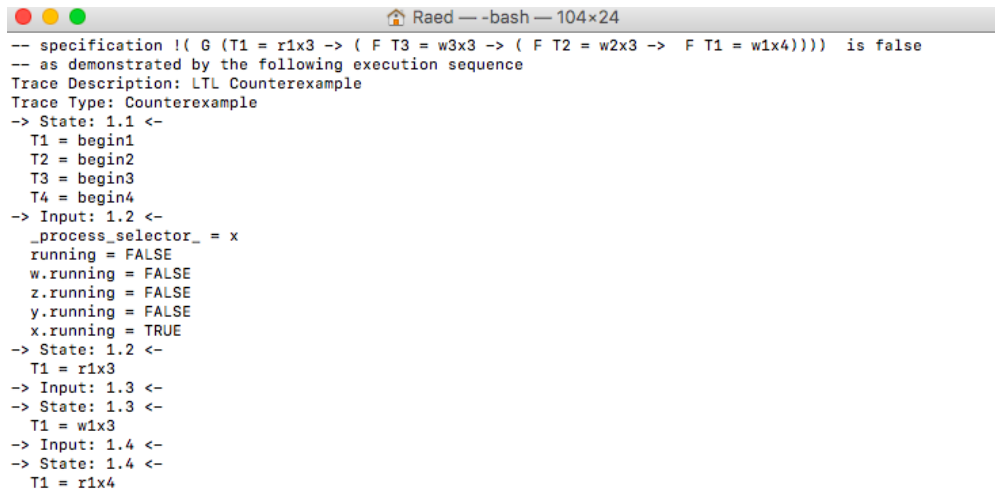
$$\sigma_6 = \bigwedge_{\substack{i,j,k \geq 1 \\ i \neq j \neq k}} \bigwedge_{\substack{x,y \in D_i \\ z \in D_j, D_k \\ y \in D_k, D_i}} ! G [(r_i(x) \vee w_i(x)) \Rightarrow F(w_j(x) \vee (w_j(x) \wedge (w_j(z) \vee r_j(z)))) \Rightarrow \\ F(w_k(x) \wedge w_k(z) \wedge w_k(y) \vee (w_k(x) \wedge (w_k(z) \wedge (w_k(y)))) \Rightarrow F(w_i(y))]$$

In this LTL formula we are looking to see if we can have a cycle of length 3. A cycle can be produced by three or more transactions where the first transaction conflicts with the second, creating an edge from the first to the second transaction in the conflict graph, and where the second transaction conflicts with the third, similarly creating an edge from the second to the third transaction in the conflict graph. Finally, the third transaction conflicts with the first, creating an edge from the third to the first transaction in the graph. In this scenario, we will have a cycle of length 3 which matches our goal. This LTL formula will look for a similar match according to the transactions available. This is encoded into LTL in NuSVM as follows:

$$\text{LTLSPEC } !G((T1=r1x3) \rightarrow F(T3=w3x3) \rightarrow F(T2=w2x3) \rightarrow F(T1=w1x4))$$

This will check whether there is no cycle of this form. This will detect the cycle  $T_1 T_3 T_2 T_1$  of length 3. As a result, the NuSMV model checker will give a counterexample stating that this condition has not been satisfied, which means that it

is not the case that we do not have this cycle, i.e., we have a cycle. The error is shown in Figure 4.3. If we remove the ! from the beginning, this will not cause an error, and instead we will have a result that is returned as being *true*.



```

Raed -- -bash -- 104x24
-- specification !( G (T1 = r1x3 -> ( F T3 = w3x3 -> ( F T2 = w2x3 -> F T1 = w1x4)))) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  T1 = begin1
  T2 = begin2
  T3 = begin3
  T4 = begin4
-> Input: 1.2 <-
  _process_selector_ = x
  running = FALSE
  w.running = FALSE
  z.running = FALSE
  y.running = FALSE
  x.running = TRUE
-> State: 1.2 <-
  T1 = r1x3
-> Input: 1.3 <-
-> State: 1.3 <-
  T1 = w1x3
-> Input: 1.4 <-
-> State: 1.4 <-
  T1 = r1x4

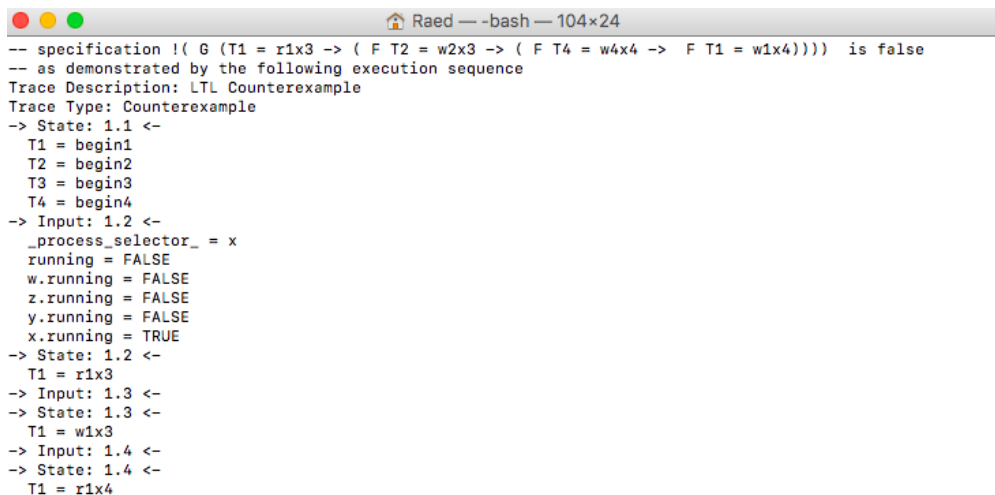
```

Figure 4.3: Counterexample on cycle  $T_1T_3T_2T_1$ .

The following LTL formula in NuSMV will check if there is a cycle of length 3 formed as  $T_1T_2T_4T_1$ :

LTLSPEC !G((T1=r1x3)->F(T2=w2x3)->F(T4=w4x4)->F(T1=w1x4))

The result in Figure 4.4 indicates a counterexample, which means that this cycle exists.



```

Raed -- -bash -- 104x24
-- specification !( G (T1 = r1x3 -> ( F T2 = w2x3 -> ( F T4 = w4x4 -> F T1 = w1x4)))) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  T1 = begin1
  T2 = begin2
  T3 = begin3
  T4 = begin4
-> Input: 1.2 <-
  _process_selector_ = x
  running = FALSE
  w.running = FALSE
  z.running = FALSE
  y.running = FALSE
  x.running = TRUE
-> State: 1.2 <-
  T1 = r1x3
-> Input: 1.3 <-
-> State: 1.3 <-
  T1 = w1x3
-> Input: 1.4 <-
-> State: 1.4 <-
  T1 = r1x4

```

Figure 4.4: Counterexample on cycle  $T_1T_2T_4T_1$ .

The following will check if there is a cycle of length 3 formed as  $T_2T_1T_4T_2$ :

LTLSPEC !G((T2=r2x3)->F(T1=w1x3)->F(T4=w4x4)->F(T2=w2x4))



Figure 4.5 shows a result that indicates a counterexample, which again means that this cycle exists.

```

Raed -- -bash -- 104x24
-- specification !( G (T2 = r2x3 -> ( F T1 = w1x3 -> ( F T4 = w4x4 -> F T2 = w2x4)))) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  T1 = begin1
  T2 = begin2
  T3 = begin3
  T4 = begin4
-> Input: 1.2 <-
  _process_selector_ = x
  running = FALSE
  w.running = FALSE
  z.running = FALSE
  y.running = FALSE
  x.running = TRUE
-> State: 1.2 <-
  T1 = r1x3
-> Input: 1.3 <-
-> State: 1.3 <-
  T1 = w1x3
-> Input: 1.4 <-
-> State: 1.4 <-
  T1 = r1x4

```

Figure 4.5: Counterexample on cycle  $T_2T_1T_4T_2$ .

The following formula will check if there is a cycle of length 3 formed as  $T_3T_2T_4T_3$ :

LTLSPEC !G((T3=r3x3)->F(T2=w2x3)->F(T4=w4x4)->F(T3=w3x5))

Figure 4.6 indicates that a counterexample is given, which means that this cycle exists.

```

Raed -- -bash -- 104x24
-- specification !( G (T3 = r3x3 -> ( F T2 = w2x3 -> ( F T4 = w4x4 -> F T3 = w3x5)))) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  T1 = begin1
  T2 = begin2
  T3 = begin3
  T4 = begin4
-> Input: 1.2 <-
  _process_selector_ = x
  running = FALSE
  w.running = FALSE
  z.running = FALSE
  y.running = FALSE
  x.running = TRUE
-> State: 1.2 <-
  T1 = r1x3
-> Input: 1.3 <-
-> State: 1.3 <-
  T1 = w1x3
-> Input: 1.4 <-
-> State: 1.4 <-
  T1 = r1x4

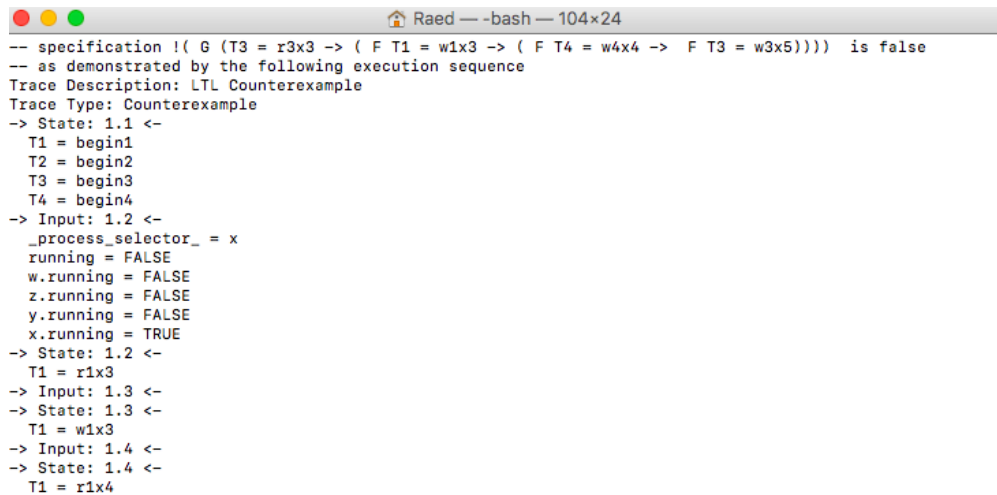
```

Figure 4.6: Counterexample on cycle  $T_3T_2T_4T_3$ .

The following formula will check if there is a cycle of length 3 formed as  $T_3T_1T_4T_3$ :

LTLSPEC !G((T3=r3x3)->F(T1=w1x3)->F(T4=w4x4)->F(T3=w3x5))

Figure 4.7 indicates that a counterexample is given, which means that this cycle exists.



```

Raed -- -bash -- 104x24
-- specification !( G (T3 = r3x3 -> ( F T1 = w1x3 -> ( F T4 = w4x4 -> F T3 = w3x5)))) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  T1 = begin1
  T2 = begin2
  T3 = begin3
  T4 = begin4
-> Input: 1.2 <-
  _process_selector_ = x
  running = FALSE
  w.running = FALSE
  z.running = FALSE
  y.running = FALSE
  x.running = TRUE
-> State: 1.2 <-
  T1 = r1x3
-> Input: 1.3 <-
-> State: 1.3 <-
  T1 = w1x3
-> Input: 1.4 <-
-> State: 1.4 <-
  T1 = r1x4

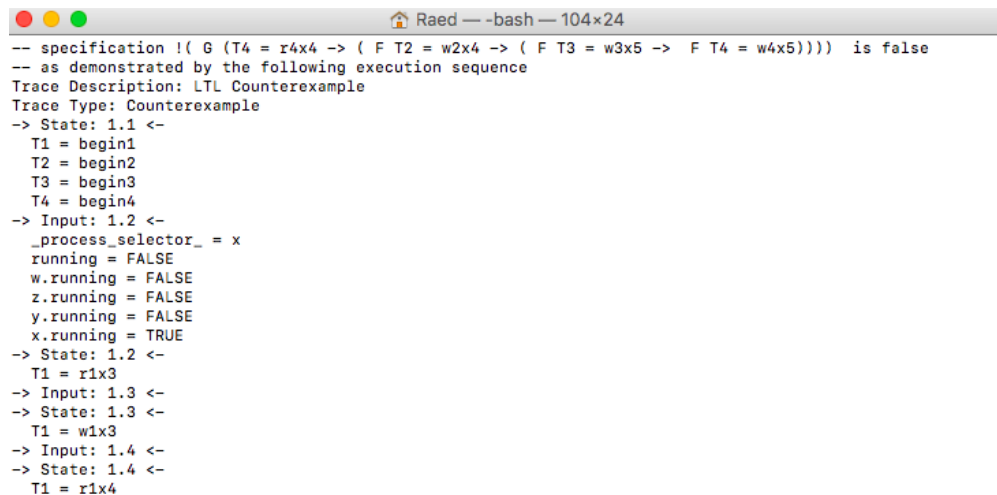
```

Figure 4.7: Counterexample on cycle  $T_3T_1T_4T_3$ .

The following formula will check if there is a cycle of length 3 formed as  $T_3T_1T_4T_3$ :

LTLSPEC !G((T4=r4x4)->F(T2=w2x4)->F(T3=w3x5)->F(T4=w4x5))

Figure 4.8 indicates that a counterexample is given, which means that this cycle exists.



```

Raed -- -bash -- 104x24
-- specification !( G (T4 = r4x4 -> ( F T2 = w2x4 -> ( F T3 = w3x5 -> F T4 = w4x5)))) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  T1 = begin1
  T2 = begin2
  T3 = begin3
  T4 = begin4
-> Input: 1.2 <-
  _process_selector_ = x
  running = FALSE
  w.running = FALSE
  z.running = FALSE
  y.running = FALSE
  x.running = TRUE
-> State: 1.2 <-
  T1 = r1x3
-> Input: 1.3 <-
-> State: 1.3 <-
  T1 = w1x3
-> Input: 1.4 <-
-> State: 1.4 <-
  T1 = r1x4

```

Figure 4.8: Counterexample on cycle  $T_3T_1T_4T_3$ .

## 4.8 Verification of the Routing Protocol using the NuSMV model checker

We will use the NuSMV model checker [80] to determine whether the protocol specifications expressed in LTL hold, or otherwise. The model checker will return a *true* result if the specification of the required property conforms with all system behaviours; otherwise, a counterexample will be given by NuSMV that represents

the error source. Encoding the protocol in LTL is shown in Appendix A. We will first explain some keywords and variables used in the model. We used `MODULE move(Tr,n,Ta,Tb,Tc)`. The variables  $(Tr,n,Ta,Tb,Tc)$  represent:

`Tr`: a transaction that is currently in process.

`n`: an integer indicating the number of the transaction.

`Ta, Tb, Tc`: other transactions that are waiting in the queue.

`T1,T2,T3, T4`: transactions number one, two, three and four.

`r1x1`: T1 reads item x1.

`w1x1`: T1 writes on item x1.

## 4.9 Conclusion

In this chapter, we have presented a protocol to be used in routing administration. The importance of this work is in the different ways routers can be accessed by an unlimited number of concurrent transactions, and how correctness (serializability) can be proved when there are gaps in successive accesses to routers using the results in [77]. We have given the specification and verification of the protocol using LTL, which was then coded into the NuSMV model checker. In order to prove serializability, so as to determine if a conflict graph contains a cycle when the number of transactions is unlimited, we computed the gaps in router accesses from which the specific length of cycle to be checked was calculated, rather than searching for cycles of all possible lengths. The anticipated benefits of this work are in the verification of administrative routing protocols. Despite the contribution of this research, and indeed its different potential applications, it is limited by the order in which the transactions have to access the data items. This opens the door to further research in this area. Future work will consider other situations where data is accessed in a different manner.

# Chapter 5

## Synchronous Network Flooding

### 5.1 Introduction

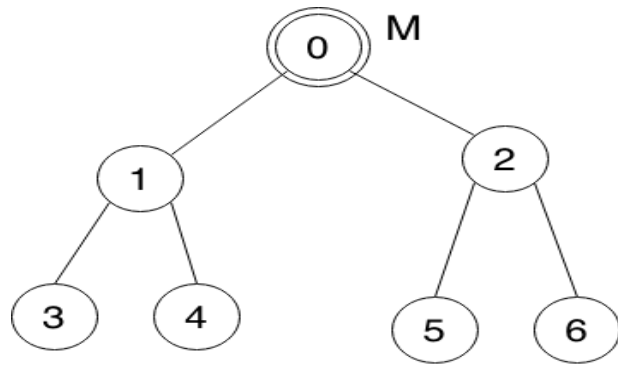
As discussed in the literature in Section 3.2, distributed systems can be specified as a composition of the specifications of their constituent components using well-studied process-calculi approaches such as CSP [1], CCS [2], the  $\pi$ -calculus [70], the Ambient Calculus [71] and I/O automata [72]. These methods are useful when components have significant internal actions/states that affect the associated external actions, but which need to be abstracted away to prove the properties of the external behaviour. Our interest in this chapter is the network flooding algorithm where individual components, in this case physical nodes in a network, have minimal internal states. The global properties to be proved derive their complexity from the topology of the network graph. It may be difficult to achieve a desired global topology as some kind of composition of components, and may necessitate an extra proof to show that the topology has indeed been achieved. We choose a more direct logic-based approach specifying the overall system - both algorithm and network topology - as a set of temporal logic constraints in order to prove the required properties [81]. This has the added benefit that a different network topology can be easily specified by changing a single constraint, rather than many components, in order to achieve the same effect.

This chapter is structured as follows. In Section 5.2, we describe the synchronous flooding algorithm. Section 5.3 then defines the temporal logic and operators used in the specification. The specification for the network flooding is given in Section 5.4, along with the proof obligation for the basic property of termination. This is applied when comparing termination in different network topologies in Section 5.5. A worked example is described in Section 5.6, as well as its proof in NuSMV. Some concluding remarks are given in Section 5.7.

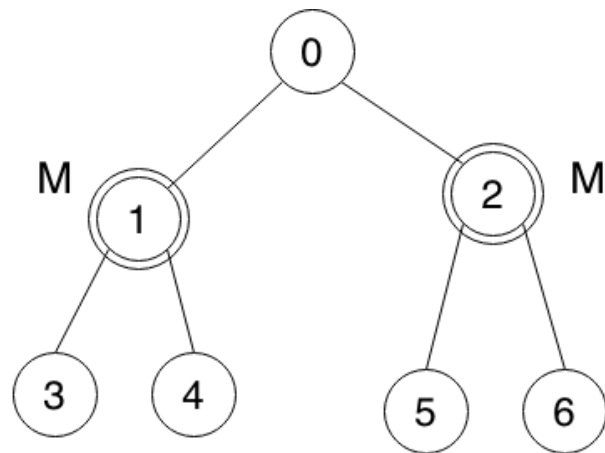
## 5.2 The Synchronous Flooding Algorithm

Distributed network routing algorithms deal with directing and redirecting messages between the different network routers and end points [17]. We refer to routers and endpoints as nodes. Synchronous distributed algorithms assume a ‘global clock’ where actions happen in clock ticks, or rounds. This means that the network has bounded link delays and lockstep synchronization with pulses of the global clock. In the message synchronization property, a message sent from node  $v$  to neighbour  $u$  at pulse  $p$  of  $v$  must be delivered to  $u$  before pulse  $p + 1$  of  $u$  [20]. In the first round, a message is sent from the initial node to its neighbours, as shown in Figure 5.1(b). In the second round, the neighbours which receive this message will forward it to all of its neighbours except the ones from which it was received. Eventually, all the nodes in the network will receive the message in a particular round.

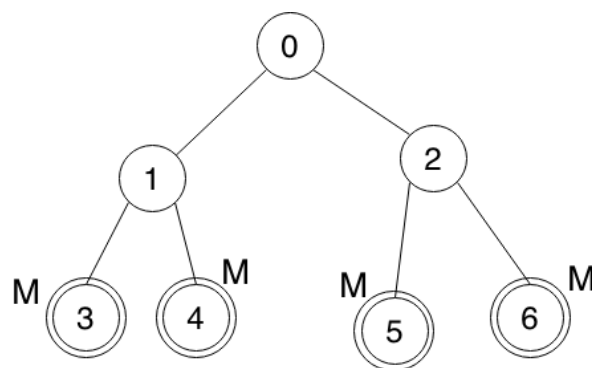
In this chapter, we investigate ‘memoryless’ flooding; that is, a node does not explicitly remember if it has previously taken part in the process or which other nodes have previously interacted with it. This may happen, for example, if the node does not have enough memory to store its past history or there are multiple flooding operations occurring simultaneously which it does not want to, or cannot, distinguish. It does, however, know which node(s) sent it the message in the present round and forwards copies of the message to all its other neighbours. Note that if in any round a node receives the message from all its neighbours, it does not need to do anything in the subsequent round. If at some point no node forwards the message we say that flooding has terminated. It is hard to be sure whether the flooding process will ever terminate, especially in complicated topologies with cycles. Figures 5.1(a)-(c) demonstrate the synchronous flooding algorithm in a network of four nodes. Nodes which hold a message “M” are double-circled. Figures 5.2 (a)-(e) demonstrates another example of the synchronous flooding algorithm in a network of three nodes; again, nodes which hold a message “M” are double-circled.



(a) Round 0



(b) Round 1



(c) Round 2

Figure 5.1: Flooding example 1

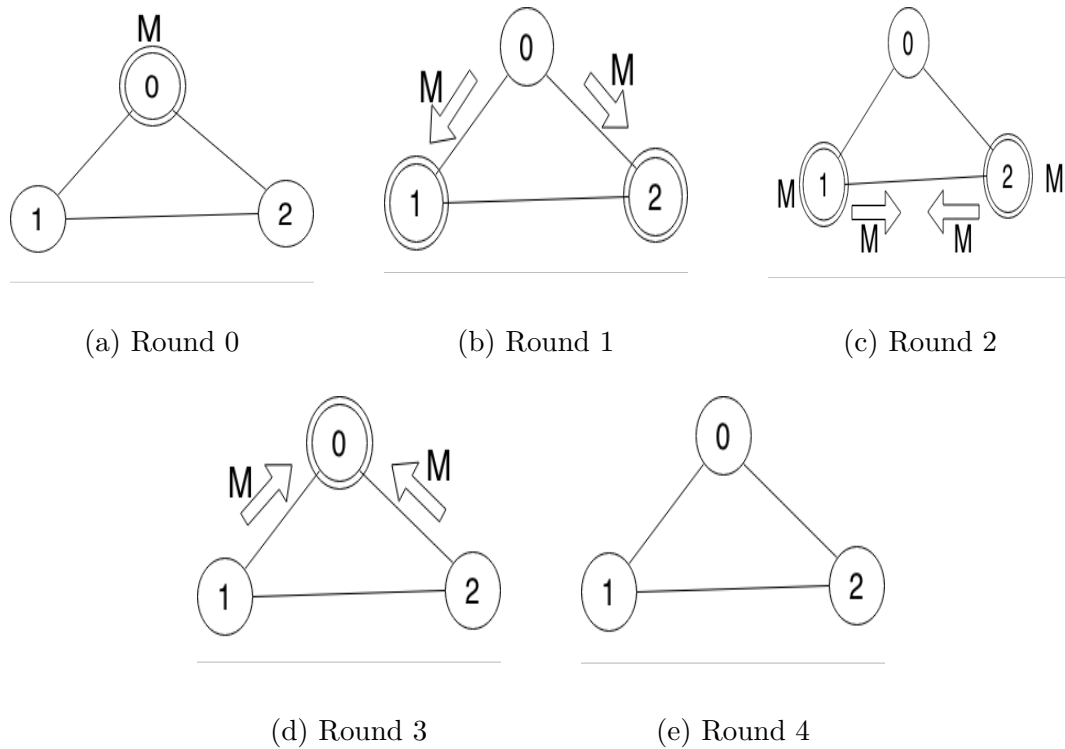


Figure 5.2: Flooding example 2

## 5.3 Linear Temporal Logic

In both this and the following chapters we will use standard Linear Temporal Logic with the temporal operators defined below.

### 5.3.1 Syntax of LTL

The LTL alphabet consists of a set of propositional symbols  $P_i$ ,  $i = 0, 1, 2, \dots$  (we will use different capital letters to  $P$  in different contexts), booleans  $\neg, \wedge, \top, \perp$ , and temporal operators  $X, Y, F, G$ . Formulae in LTL are those generated by:

$$\phi ::= P_i | \neg\phi | \phi_1 \wedge \phi_2 | X\phi | Y\phi | F\phi | G\phi$$

The Boolean connectives  $\vee, \Rightarrow$  and  $\Leftrightarrow$  will be defined in terms of  $\neg$  and  $\wedge$  in the usual manner.

### 5.3.2 Semantics of LTL

Linear Temporal Logic is interpreted over a sequence of temporal states (which we sometimes refer to as ‘points in time’, even though they may not themselves represent real time)  $s_0, \dots, s_a, \dots$  ( $a \in \mathbb{N}$ ). An interpretation of LTL,  $I(s_a)$ , at a given state  $s_a$  assigns truth values  $P_i^{I(s_a)}$  to the propositional symbols  $P_i$ . A

structure  $M$  is a sequence of interpretations  $I(s_0), \dots, I(s_a), \dots$  for the sequence of states. The semantics of the LTL formula  $\phi$  are given by a truth relationship  $M, s_a \models \phi$ , which means that  $\phi$  holds at state  $s_a$  in the structure  $M$ . The relation  $\models$  is defined inductively as follows:

$$\begin{aligned}
M, s_a \models P_i &\text{ iff } p_i^{I(s_a)} = \top \\
M, s_a \models \neg\phi &\text{ iff } M, s_a \not\models \phi \\
M, s_a \models \phi_1 \wedge \phi_2 &\text{ iff } M, s_a \models \phi_1 \text{ and } M, s_a \models \phi_2 \\
M, s_a \models \mathbf{X}\phi &\text{ iff } M, s_{a+1} \models \phi \\
M, s_a \models \mathbf{Y}\phi &\text{ iff } M, s_{a-1} \models \phi \quad (a > 0) \\
M, s_a \models \mathbf{F}\phi &\text{ iff there exists } k \geq a \text{ such that } M, s_k \models \phi \\
M, s_a \models \mathbf{G}\phi &\text{ iff, for all } k \geq a, \quad M, s_k \models \phi
\end{aligned}$$

Intuitively, the temporal operator X reads as “in the next state”, Y reads as “in the previous state”, F reads as “in some future state”, and G reads as “in all future states”. A structure  $M$  is a *model* of an LTL formula  $\phi$  if

$$M, s_0 \models \phi$$

In general, a given LTL formula  $\phi$  will have many models. The behaviour of network flooding in different contexts (e.g., in different network topologies) is a set of models. We specify such network flooding in temporal logic by giving a LTL formula  $\phi$  whose models correspond exactly to the behaviour exhibited by network flooding. We can then use this  $\phi$  to construct further LTL formulae (called ‘proof obligations’) that assert the properties of  $\phi$  such as when flooding behaviour leads to termination.

## 5.4 Specification of the Synchronous Flooding Algorithm

In the specification here, temporal logic states will correspond to rounds in the progression of the network flooding. Successive rounds change the state of the network; for example, the nodes in the network that are receiving messages in a particular round. Let  $N$  be the set of nodes in the size of the network under consideration. The following subsections give the propositions and constraints on these nodes that define the behaviour of the rounds.



### 5.4.1 Edge Propositions

The set of graph edge propositions is given by:

$$\{E_{\{g,h\}} \mid g, h \in N, g \neq h\}$$

Intuitively,  $E_{\{g,h\}}$  is true if there is an edge between the distinct nodes  $g$  and  $h$  in the graph. Note that we have used a set  $\{g, h\}$  as a subscript in  $E_{\{g,h\}}$ , which is to indicate that  $E_{\{g,h\}}$  is the same proposition as  $E_{\{h,g\}}$ , i.e., edges in  $N$  are undirected, and to say that an edge from  $g$  to  $h$  is the same as saying that there is an edge from  $h$  to  $g$ . We can specify whether two nodes  $g$  and  $h$  have an edge between them by specifying whether  $E_{\{g,h\}}$  is *True* or *False*. In this way, a specific graph topology can be defined one edge at a time. Secondly, we can give general Boolean constraints on the edge propositions. The set of solutions for the constraints is the set of combinations of the edge propositions that are *True*, corresponding to a set of graph topologies for  $N$ . As a third possibility, we may choose not to specify any Boolean constraints on edge propositions if we want to prove some particular network flooding property for all graph topologies on  $N$ . However, the use of edge propositions does require a basic *temporal* constraint, namely that the Boolean value of an edge variable is time-independent. Nodes  $g$  and  $h$  have an edge between them either always or never, as edges represent physical connections that do not change with time. This temporal constraint is given by:

$$\phi_e \equiv \bigwedge_{\substack{g,h \in N, \\ g \neq h}} (\mathbf{G}E_{\{g,h\}} \vee \mathbf{G}\neg E_{\{g,h\}})$$

### 5.4.2 Send-message Propositions

Messages may be sent between nodes  $g$  and  $h$  in both directions. Thus, we have the send propositions

$$\{S_{g,h} \mid g, h \in N, g \neq h\}$$

where  $S_{g,h}$  is true in a particular round if node  $g$  sends a message to node  $h$  in that same round. As sending messages between nodes is directional,  $S_{g,h}$  and  $S_{h,g}$  are different propositions which may differ on their respective truth values in each round. Also, the sending of messages is time-dependent, so the truth value of a particular send will vary over time. The basic constraint on send propositions relates to the edge propositions, as messages can only be sent from node  $g$  to node  $h$  along an edge from  $g$  to  $h$ , and accordingly  $E_{\{g,h\}}$  has to be *True*. The

constraint is:

$$\phi_s \equiv \bigwedge_{\substack{g,h \in N, \\ g \neq h}} \mathbf{G}(S_{g,h} \Rightarrow E_{\{g,h\}})$$

This states that, at any given point in time, if a message is sent from node  $g$  to node  $h$ , i.e., that  $S_{g,h}$  is *True*, then there must be an edge between  $g$  and  $h$  at that point in time, i.e.,  $E_{\{g,h\}}$  is *True*. This constraint and, indeed, edge propositions in general are only needed when a class of graph topologies for  $N$  is being considered where edges may be present in particular topologies in the class but absent in others. If a fixed graph topology is under consideration, we do not need edge propositions as we can explicitly restrict the set of send propositions to pairs of nodes between which we know edges exist.

### 5.4.3 Message-received Propositions

We have a set of propositions  $M_g$  for the nodes  $g \in N$

$$\{M_g \mid g \in N\}$$

such that, in any given round,  $M_g$  is *True* if node  $g$  receives a message. In our model of the flooding algorithm, after the initial round, node  $g$  holds a message if it has been received by  $g$  in that same round, i.e., some neighbour node  $h$  sends a message to  $g$  in that round - see the first conjunct in  $\phi_m$  below. However, node  $h$  will only send a message to  $g$  if  $g$  did not send a message to  $h$  in the previous round - see the second conjunct in  $\phi_m$  below.

$$\phi_m \equiv (\mathbf{XG} \bigwedge_{g \in N} (M_g \Leftrightarrow \bigvee_{\substack{h \in N, \\ h \neq g}} S_{h,g})) \wedge (\mathbf{XG} \bigwedge_{\substack{g,h \in N, \\ g \neq h}} (S_{g,h} \Leftrightarrow \mathbf{Y}(M_g \wedge \neg S_{h,g})))$$

### 5.4.4 Initial Conditions

The initial temporal state corresponds to the initial round when some initial node holds a message which is then sent to all its neighbours in the next round, thus triggering network flooding. Therefore,  $M_g$  will be true for exactly one  $i_0 \in N$  and, as our send-message propositions are *True* in the round that the corresponding message is received, no send-message proposition is *True* in the initial round. These two conditions are captured in the two outer-level conjuncts below:

$$\phi_i \equiv (M_{i_0} \wedge \bigwedge_{g \in N, g \neq i_0} \neg M_g) \wedge (\bigwedge_{g,h \in N} \neg S_{g,h})$$

If we want to vary the initial node, we can use the following variable version:

$$\phi_{i_v} \equiv \left( \bigvee_{i \in M} M_i \wedge \bigwedge_{g \in N, g \neq i} \neg M_g \right) \wedge \left( \bigwedge_{g, h \in N} \neg S_{g, h} \right)$$

### 5.4.5 Topological Constraints

In 4.1, we stated that edges of  $N$  can be defined in one of three ways:

- (i) explicitly define a single topology for  $N$  by listing the edges;
- (ii) implicitly define a class of topologies for  $N$  by defining the constraints on edges in  $N$ ;
- (iii) allow for all topologies in  $N$ .

Case (iii) means that there are no constraints. We give a worked example of case (i) later in the chapter. Here, we consider case (ii), and show how common classes of network topologies that are of interest in network flooding can be defined by Boolean constraints on the propositions  $E_{\{g, h\}}(g, h \in N)$ .

#### 5.4.5.1 Regular Graphs

Suppose that  $N$  has  $n$  nodes:

$$N = \{g_1, \dots, g_n\}$$

A *regular* graph with nodes  $N$  has degree  $m$ , where  $1 \leq m \leq n-1$ , i.e., every node  $g \in N$  has  $m$  neighbours. The class of all regular topologies in  $N$  is specified by the following condition on the edge propositions below. We denote the cardinality of a set  $H$  by  $|H|$ .

$$\phi_{top} \equiv \bigvee_{1 \leq m < n-1} \bigwedge_{g \in N} \bigvee_{\substack{H \subseteq N - \{g\}, \\ |H|=m}} \left( \bigwedge_{h \in H} E_{\{g, h\}} \wedge \bigwedge_{h \notin H} \neg E_{\{g, h\}} \right)$$

For the set of nodes  $N = \{1, 2, 3, 4\}$ ,  $\phi_{top}$  instantiates to:

$$\begin{aligned}
& ( ((E_{\{1,2\}} \wedge \neg E_{\{1,3\}} \wedge \neg E_{\{1,4\}}) \vee (E_{\{1,3\}} \wedge \neg E_{\{1,2\}} \wedge \neg E_{\{1,4\}}) \vee (E_{\{1,4\}} \wedge \neg E_{\{1,2\}} \wedge \neg E_{\{1,3\}})) \\
& \wedge ((E_{\{2,1\}} \wedge \neg E_{\{2,3\}} \wedge \neg E_{\{2,4\}}) \vee (E_{\{2,3\}} \wedge \neg E_{\{2,1\}} \wedge \neg E_{\{2,4\}}) \vee (E_{\{2,4\}} \wedge \neg E_{\{2,1\}} \wedge \neg E_{\{2,3\}})) \\
& \wedge ((E_{\{3,1\}} \wedge \neg E_{\{3,2\}} \wedge \neg E_{\{3,4\}}) \vee (E_{\{3,2\}} \wedge \neg E_{\{3,1\}} \wedge \neg E_{\{3,4\}}) \vee (E_{\{3,4\}} \wedge \neg E_{\{3,1\}} \wedge \neg E_{\{3,2\}})) \\
& \wedge ((E_{\{4,1\}} \wedge \neg E_{\{4,2\}} \wedge \neg E_{\{4,3\}}) \vee (E_{\{4,2\}} \wedge \neg E_{\{4,1\}} \wedge \neg E_{\{4,3\}}) \vee (E_{\{4,3\}} \wedge \neg E_{\{4,1\}} \wedge \neg E_{\{4,2\}})) ) \\
& \vee \\
& ( ((E_{\{1,2\}} \wedge E_{\{1,3\}} \wedge \neg E_{\{1,4\}}) \vee (E_{\{1,3\}} \wedge \neg E_{\{1,2\}} \wedge E_{\{1,4\}}) \vee (E_{\{1,4\}} \wedge E_{\{1,2\}} \wedge \neg E_{\{1,3\}})) \\
& \wedge ((E_{\{2,1\}} \wedge E_{\{2,3\}} \wedge \neg E_{\{2,4\}}) \vee (E_{\{2,3\}} \wedge \neg E_{\{2,1\}} \wedge E_{\{2,4\}}) \vee (E_{\{2,4\}} \wedge E_{\{2,1\}} \wedge \neg E_{\{2,3\}})) \\
& \wedge ((E_{\{3,1\}} \wedge E_{\{3,2\}} \wedge \neg E_{\{3,4\}}) \vee (E_{\{3,2\}} \wedge \neg E_{\{3,1\}} \wedge E_{\{3,4\}}) \vee (E_{\{3,4\}} \wedge E_{\{3,1\}} \wedge \neg E_{\{3,2\}})) \\
& \wedge ((E_{\{4,1\}} \wedge E_{\{4,2\}} \wedge \neg E_{\{4,3\}}) \vee (E_{\{4,2\}} \wedge \neg E_{\{4,1\}} \wedge E_{\{4,3\}}) \vee (E_{\{4,3\}} \wedge E_{\{4,1\}} \wedge \neg E_{\{4,2\}})) )
\end{aligned}$$

### 5.4.5.2 Expander Graphs

Expanders are a very important class of graphs (having the property of being simultaneously sparse and well connected) that have applications in various areas of computer science and mathematics; for instance, in the design and analysis of communication networks, cryptography, error-correcting codes, pseudorandomness, complexity, coding theory, metric embeddings, etc. (for details, see this well-known survey [56]). For example, in the context of distributed computer networks they have been used for building censorship-resistant networks [57, 58], fault tolerant networks [59], efficient (Byzantine) agreement and leader election algorithms [60, 61, 62, 63], analysing information spreading, etc. [64]. Thus, even the efficient construction (in static or dynamic fault-tolerant settings) of expander networks is an important line of research [65, 66, 67, 68, 69].

Intuitively, an ‘expander’ graph  $N$  is one where every subset  $S \subseteq N$  of vertices expands ‘quickly’; how quickly it expands is determined by an ‘expansion parameter’. A graph  $N$  has *expansion parameter*  $\epsilon$  if, for every subset  $S \subseteq N$  with  $|S| \leq |N|/2$ , the set of edges connecting nodes in  $S$  with nodes not in  $S$  is greater than or equal to  $\epsilon|S|$ . We can constrain the network  $N$  to topologies with the expansion parameter  $\epsilon$  by the following Boolean constraint on propositions:

$$\phi_{top} \equiv \bigwedge_{\substack{S \subseteq N, \\ |S| \leq |N|/2}} \bigvee_{\substack{T \subseteq N \times N, \\ |T| \geq \epsilon|S|}} \bigwedge_{\{g,h\} \in T} (E_{\{g,h\}} \wedge (\{g,h\} \cup S \neq \emptyset) \wedge (\{g,h\} \not\subseteq S))$$

Here, the set  $N \times N$  is the set of ordered pairs of nodes  $(g, h)$  in the Cartesian product  $N \times N$  viewed as two-element sets  $\{g, h\}$  (so that  $\{g, h\} = \{h, g\}$ , whereas  $(g, h) \neq (h, g)$ ). Also,  $\{g, h\} \cup S \neq \emptyset$  and  $\{g, h\} \not\subseteq S$  are evaluated as being *True* or *False* accordingly in each respective conjunct. The constraint essentially states that corresponding to every subset of nodes  $S$ , with  $|S| \leq |N|/2$ , there is a set of

edges  $T$ , where  $|T| \geq \epsilon|S|$ , each of which connects a node in  $S$  with a node not in  $S$ .

### 5.4.6 Termination

The required property that first comes to mind in network flooding is termination. Termination occurs if, in some round, no node in the system receives a message. In our temporal model, this means that no message-received proposition  $m_g$  will be true. So, if network flooding is modelled by  $\phi_e$ ,  $\phi_s$ ,  $\phi_m$ ,  $\phi_i$  and  $\phi_{top}$  as above, then the proof obligation for termination is:

$$\phi_e \wedge \phi_s \wedge \phi_m \wedge \phi_i \wedge \phi_{top} \Rightarrow \mathbf{F} \bigwedge_{g \in N} \neg m_g$$

## 5.5 Applications

We use our specification of flooding to compare the time it takes for the flooding algorithm to terminate in different topologies. Whilst standard LTL is not designed to resolve timing issues, we can determine which network topology takes fewer rounds to terminate by superimposing the temporal behaviour of the network in one topology on the behaviour of another. So, the temporal model has two cases of network flooding, on the same set of nodes but with different connections and proceeding together in rounds in a lock-step fashion, and with two messages - one for each topological case - circulating in the network. We can illustrate this model with a simple example. Suppose that  $N = \{0, 1, 2\}$  and the two topologies are constructed as shown in the following figure:



Figure 5.3: Flooding on two topologies

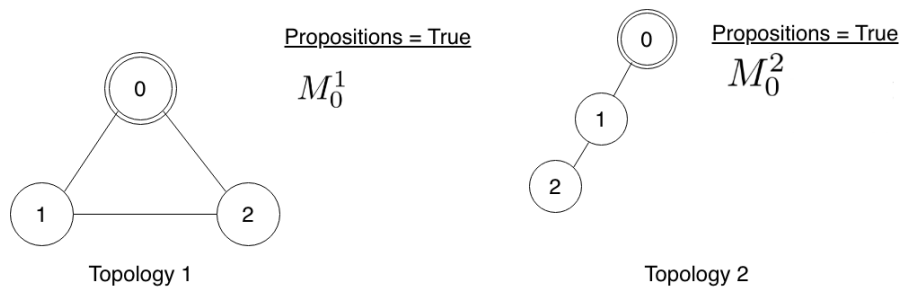
Assume 0 is the initial node in both cases. We illustrate the progression of the rounds in terms of the send-message propositions  $S_{g,h}$  and message-received

propositions  $M_g$ . Distinguishing these propositions for the two topologies, we have the following propositions:

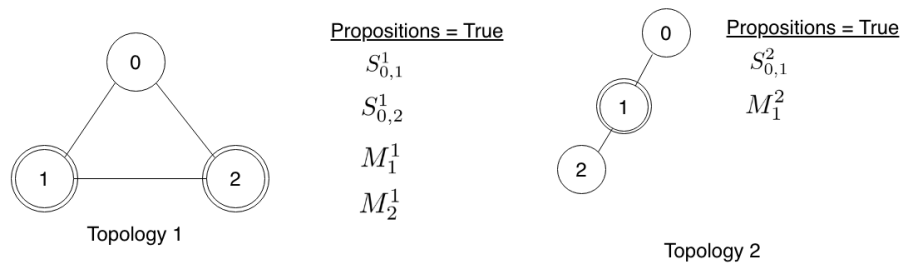
$$\textit{Topology1} : S_{0,1}^1, S_{0,2}^1, S_{1,2}^1, M_0^1, M_1^1, M_2^1$$

$$\textit{Topology2} : S_{0,1}^2, S_{1,2}^2, M_0^2, M_1^2, M_2^2$$

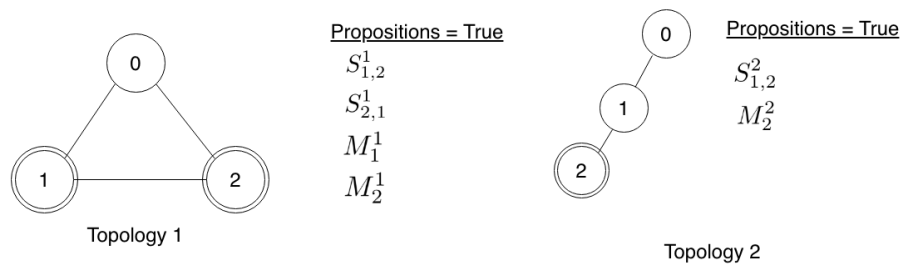
The propositions that are *True* in successive rounds in the two models are shown in Figures 5.4(a)-(d) below. Nodes which hold a message are circled.



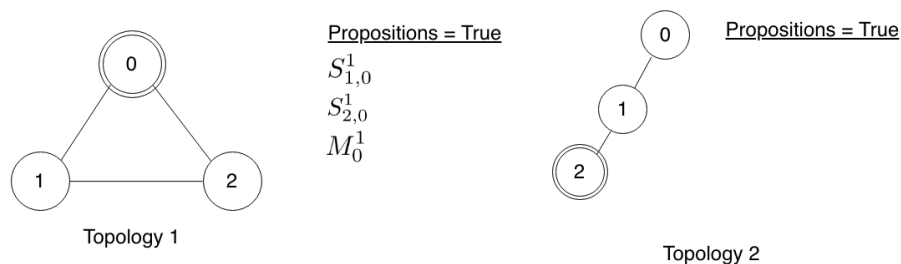
(a) Round 0



(b) Round 1



(c) Round 2



(d) Round 3

Figure 5.4: Flooding rounds in two topologies

Note that in round 3  $M_0^1$  is *True*, whereas  $M_0^2$ ,  $M_1^2$ , or  $M_2^2$  are not *True*. So, *Topology2* terminates before *Topology1* as there is a round in which no node holds a message in *Topology2*. whereas a node still holds a message in *Topology1*. We

give the formal proof that has to be carried out in the general case below.

Given two topologies *Topology1* and *Topology2* on a set of nodes  $N$ , the proof obligation that *Topology1* terminates before *Topology2* is:

$$\begin{aligned}
 & (\phi_e^1 \wedge \phi_s^1 \wedge \phi_m^1 \wedge \phi_i^1 \wedge \phi_{top}^1) \wedge (\phi_e^2 \wedge \phi_s^2 \wedge \phi_m^2 \wedge \phi_i^2 \wedge \phi_{top}^2) \Rightarrow \\
 & \mathbf{F}((\bigwedge_{g \in N} \neg M_g^1) \wedge (\bigvee_{g \in N} M_g^2))
 \end{aligned} \tag{5.1}$$

Here,  $\phi_e^1$ ,  $\phi_s^1$  and  $\phi_m^1$  relabel the propositional variables from  $\phi_e$ ,  $\phi_s$  and  $\phi_m$  of subsections 5.4.1, 5.4.2, 5.4.3 respectively, adding a superscript 1, whilst  $\phi_e^2$ ,  $\phi_s^2$  and  $\phi_m^2$  do the same but with a superscript 2. The formulae  $\phi_i^1$  and  $\phi_i^2$  also possibly differ in their respective initial nodes, and  $\phi_{top}^1$  and  $\phi_{top}^2$  according to the topologies that they define. In (5.1), we check for validity. So, if  $\phi_{top}^1$  and  $\phi_{top}^2$  each define a range of topologies, (5.1) is *True* (valid) if all topologies of  $\phi_{top}^1$  terminate before all topologies of  $\phi_{top}^2$ . If (5.1) returns *False*, then some topology of  $\phi_{top}^2$  terminates before some topology of  $\phi_{top}^1$ . We could then proceed to test if all the topologies of  $\phi_{top}^2$  terminate before all those of  $\phi_{top}^1$  by checking the validity of:

$$\begin{aligned}
 & (\phi_e^2 \wedge \phi_s^2 \wedge \phi_m^2 \wedge \phi_i^2 \wedge \phi_{top}^2) \wedge (\phi_e^1 \wedge \phi_s^1 \wedge \phi_m^1 \wedge \phi_i^1 \wedge \phi_{top}^1) \Rightarrow \\
 & \mathbf{F}((\bigwedge_{g \in N} \neg M_g^2) \wedge (\bigvee_{g \in N} M_g^1))
 \end{aligned} \tag{5.2}$$

It is possible that (5.2) would also return *False*, in which case some topologies of  $\phi_{top}^1$  would terminate before some topologies of  $\phi_{top}^2$ , and further that some topologies of  $\phi_{top}^2$  would terminate before some topologies of  $\phi_{top}^1$ .

Apart from varying topologies of the network  $N$ , we could also vary the initial node. This can be achieved by replacing the initial conditions  $\phi_i$  that have a fixed initial node, by initial conditions  $\phi_{i_v}$  that vary the initial node, in the proof obligation. Thus,

$$\begin{aligned}
 & (\phi_e^1 \wedge \phi_s^1 \wedge \phi_m^1 \wedge \phi_{i_v}^1 \wedge \phi_{top}^1) \wedge (\phi_e^2 \wedge \phi_s^2 \wedge \phi_m^2 \wedge \phi_{i_v}^2 \wedge \phi_{top}^2) \Rightarrow \\
 & \mathbf{F}((\bigwedge_{g \in N} \neg M_g^1) \wedge (\bigvee_{g \in N} M_g^2))
 \end{aligned} \tag{5.3}$$

is valid if, for all topologies  $\phi_{top}^1$  starting from any initial node, flooding terminates before flooding terminates in any topology  $\phi_{top}^2$  with any initial node.



## 5.6 Worked Example

Here, we compare the termination of two topologies on a network of five nodes through the use of formal proofs. The two network topologies for this example are depicted in Figure 5.5.

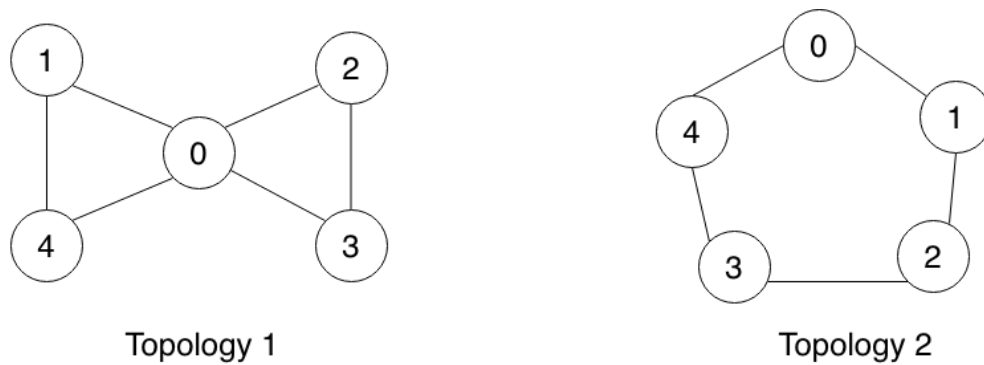


Figure 5.5: Two topologies containing five nodes

To check if a property is met, after creating the system model and its properties we encode it into the model checker NuSMV to verify if the property is satisfied. Synchronous flooding block diagram representing this is shown below in Figure 5.6.

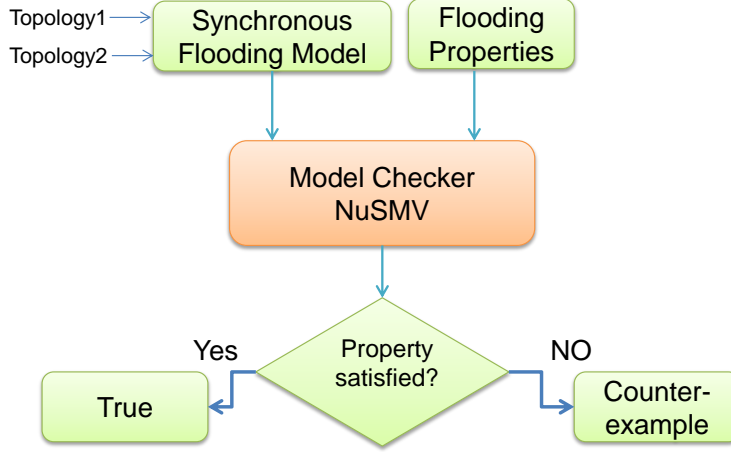


Figure 5.6: Synchronous flooding model checking block diagram.

Firstly, we will test to see if one topology terminates before the other, where both have the initial node 0. As mentioned in 5.4.2, we may optimize the number of propositions used by only having send-message propositions for the edges that are present in each of the respective topologies, which is valid in this instance as we are comparing two fixed topologies. This restriction on send-message variables for each topology also defines the topology, and thus no additional variable edge propositions  $E_{i,j}$  are required. Thus, we have the following propositions for the two topologies in Figure 5.5 above:

$$\begin{aligned}
 \textit{Topology1} : & S_{0,1}^1, S_{1,0}^1, S_{0,2}^1, S_{2,0}^1, S_{0,3}^1, S_{3,0}^1, \\
 & S_{0,4}^1, S_{4,0}^1, S_{1,4}^1, S_{4,1}^1, S_{2,3}^1, S_{3,2}^1, \\
 & M_0^1, M_1^1, M_2^1, M_3^1, M_4^1 \\
 \textit{Topology2} : & S_{0,1}^2, S_{1,0}^2, S_{1,2}^2, S_{2,1}^2, S_{2,3}^2, S_{3,2}^2, S_{3,4}^2, S_{4,3}^2, S_{0,4}^2, S_{4,0}^2, \\
 & M_0^2, M_2^2, M_2^2, M_3^2, M_4^2
 \end{aligned}$$

As there are no edge propositions, we ignore  $\phi_e$ ,  $\phi_s$  and  $\phi_{top}$  and only consider  $\phi_i$  and  $\phi_m$  for each topology. Instantiating the definitions of  $\phi_i$  and  $\phi_m$  of subsections

5.4.4 and 5.4.3 respectively, yields:

$$\begin{aligned}\phi_i^1 \equiv & M_0^1 \wedge \neg M_1^1 \wedge \neg M_2^1 \wedge \neg M_3^1 \wedge \neg M_4^1 \wedge \\ & \neg(S_{0,1}^1 \vee S_{1,0}^1 \vee S_{0,2}^1 \vee S_{2,0}^1 \vee S_{0,3}^1 \vee S_{3,0}^1 \vee \\ & S_{0,4}^1 \vee S_{4,0}^1 \vee S_{1,4}^1 \vee S_{4,1}^1 \vee S_{2,3}^1 \vee S_{3,2}^1)\end{aligned}$$

$$\begin{aligned}\phi_m^1 \equiv & (\mathbf{XG}( (M_0^1 \Leftrightarrow S_{1,0}^1 \vee S_{2,0}^1 \vee S_{3,0}^1 \vee S_{4,0}^1) \wedge \\ & (M_1^1 \Leftrightarrow S_{0,1}^1 \vee S_{4,1}^1) \wedge \\ & (M_2^1 \Leftrightarrow S_{0,2}^1 \vee S_{3,2}^1) \wedge \\ & (M_3^1 \Leftrightarrow S_{0,3}^1 \vee S_{2,3}^1) \wedge \\ & (M_4^1 \Leftrightarrow S_{0,4}^1 \vee S_{1,4}^1) ) \wedge \\ & (\mathbf{XG}( (S_{0,1}^1 \Leftrightarrow \mathbf{Y}(M_0^1 \wedge \neg S_{1,0}^1)) \wedge \\ & (S_{1,0}^1 \Leftrightarrow \mathbf{Y}(M_1^1 \wedge \neg S_{0,1}^1)) \wedge \\ & (S_{2,0}^1 \Leftrightarrow \mathbf{Y}(M_2^1 \wedge \neg S_{0,2}^1)) \wedge \\ & (S_{0,2}^1 \Leftrightarrow \mathbf{Y}(M_0^1 \wedge \neg S_{2,0}^1)) \wedge \\ & (S_{0,3}^1 \Leftrightarrow \mathbf{Y}(M_0^1 \wedge \neg S_{3,0}^1)) \wedge \\ & (S_{3,0}^1 \Leftrightarrow \mathbf{Y}(M_3^1 \wedge \neg S_{0,3}^1)) \wedge \\ & (S_{0,4}^1 \Leftrightarrow \mathbf{Y}(M_0^1 \wedge \neg S_{4,0}^1)) \wedge \\ & (S_{4,0}^1 \Leftrightarrow \mathbf{Y}(M_4^1 \wedge \neg S_{0,4}^1)) \wedge \\ & (S_{1,4}^1 \Leftrightarrow \mathbf{Y}(M_1^1 \wedge \neg S_{4,1}^1)) \wedge \\ & (S_{4,1}^1 \Leftrightarrow \mathbf{Y}(M_4^1 \wedge \neg S_{1,4}^1)) \wedge \\ & (S_{2,3}^1 \Leftrightarrow \mathbf{Y}(M_2^1 \wedge \neg S_{3,2}^1)) \wedge \\ & (S_{3,2}^1 \Leftrightarrow \mathbf{Y}(M_3^1 \wedge \neg S_{2,3}^1))) )\end{aligned}$$

$$\begin{aligned}\phi_i^2 \equiv & M_0^2 \wedge \neg M_1^2 \wedge \neg M_2^2 \wedge \neg M_3^2 \wedge \neg M_4^2 \wedge \\ & \neg(S_{0,1}^2 \vee S_{1,0}^2 \vee S_{1,2}^2 \vee S_{2,1}^2 \vee \\ & S_{2,3}^2 \vee S_{3,2}^2 \vee S_{3,4}^2 \vee S_{4,3}^2 \vee S_{0,4}^2 \vee S_{4,0}^2)\end{aligned}$$

$$\begin{aligned}
\phi_m^2 \equiv & (\mathbf{XG}( (M_0^2 \Leftrightarrow S_{1,0}^2 \vee S_{4,0}^2) \wedge \\
& (M_1^2 \Leftrightarrow S_{0,1}^2 \vee S_{2,1}^2) \wedge \\
& (M_2^2 \Leftrightarrow S_{1,2}^2 \vee S_{3,2}^2) \wedge \\
& (M_3^2 \Leftrightarrow S_{2,3}^2 \vee S_{4,3}^2) \wedge \\
& (M_4^2 \Leftrightarrow S_{3,4}^2 \vee S_{0,4}^2) ) ) \wedge \\
& (\mathbf{XG}( (S_{0,1}^2 \Leftrightarrow \mathbf{Y}(M_0^2 \wedge \neg S_{1,0}^2)) \wedge \\
& (S_{1,0}^2 \Leftrightarrow \mathbf{Y}(M_1^2 \wedge \neg S_{0,1}^2)) \wedge \\
& (S_{1,2}^2 \Leftrightarrow \mathbf{Y}(M_1^2 \wedge \neg S_{2,1}^2)) \wedge \\
& (S_{2,1}^2 \Leftrightarrow \mathbf{Y}(M_2^2 \wedge \neg S_{1,2}^2)) \wedge \\
& (S_{2,3}^2 \Leftrightarrow \mathbf{Y}(M_2^2 \wedge \neg S_{3,2}^2)) \wedge \\
& (S_{3,2}^2 \Leftrightarrow \mathbf{Y}(M_3^2 \wedge \neg S_{2,3}^2)) \wedge \\
& (S_{3,4}^2 \Leftrightarrow \mathbf{Y}(M_3^2 \wedge \neg S_{4,3}^2)) \wedge \\
& (S_{4,3}^2 \Leftrightarrow \mathbf{Y}(M_4^2 \wedge \neg S_{3,4}^2)) \wedge \\
& (S_{4,0}^2 \Leftrightarrow \mathbf{Y}(M_4^2 \wedge \neg S_{0,4}^2)) \wedge \\
& (S_{0,4}^2 \Leftrightarrow \mathbf{Y}(M_0^2 \wedge \neg S_{4,0}^2)) ) )
\end{aligned}$$

To prove that *Topology1* terminates before *Topology2* when the initial node is selected to be node 0, we need to prove (by (5.2) in section 5.4.5 above, ignoring  $\phi_e$ ,  $\phi_s$ , and  $\phi_{top}$ ) that:

$$\begin{aligned}
& (\phi_m^1 \wedge \phi_i^1) \wedge (\phi_m^2 \wedge \phi_i^2) \Rightarrow \\
& \mathbf{F}((\neg M_0^1 \wedge \neg M_1^1 \wedge \neg M_2^1 \wedge \neg M_3^1 \wedge \neg M_4^1) \wedge (M_0^2 \vee M_1^2 \vee M_2^2 \vee M_3^2 \vee M_4^2))
\end{aligned}$$

This proof has been carried out using NuSMV and does indeed return *True*, showing that *Topology1* terminates before *Topology2* when the initial node is selected to be node 0 for both. Appendix B Section B.2 shows encoding this case in LTL into the NuSMV model checker. Figure 5.7 shows this result.

```

*** This is NuSMV 2.5.4 (compiled on Fri Nov 23 21:36:06 UTC 2012)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>

*** Copyright (c) 2010, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification F ((((!n1_0.has_message & !n1_1.has_message) & !n1_2.has_message) &
!n1_3.has_message) & !n1_4.has_message) & (((n2_0.has_message | n2_1.has_message) |
n2_2.has_message) | n2_3.has_message) | n2_4.has_message)) is true
eduroam-visitor-pt1-175-190:~ Raed$ █

```

Figure 5.7: Toplogy1 terminates before Topology2 with initial node 0.

To verify our specification, we have also used NuSMV to demonstrate that the following expression, which states that *Topology2* terminates before *Topology1* when the initial node is selected to be node 0 for both:

$$(\phi_m^2 \wedge \phi_i^2) \wedge (\phi_m^1 \wedge \phi_i^1)$$

$$\Rightarrow \mathbf{F}((\neg M_0^2 \wedge \neg M_1^2 \wedge \neg M_2^2 \wedge \neg M_3^2 \wedge \neg M_4^2) \wedge (M_0^1 \vee M_1^1 \vee M_2^1 \vee M_3^1 \vee M_4^1))$$

is *False*. Appendix B Section B.1 shows encoding this case in LTL into the NuSMV model checker. Indeed, NuSMV does return *False* as shown in Figure 5.8.

---

```

*** This is NuSMV 2.5.4 (compiled on Fri Nov 23 21:36:06 UTC 2012)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>

*** Copyright (c) 2010, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification F (((!(n2_0.has_message & !n2_1.has_message) & !n2_2.has_message) &
!n2_3.has_message) & !n2_4.has_message) & (((n1_0.has_message | n1_1.has_message) |
n1_2.has_message) | n1_3.has_message) | n1_4.has_message) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  n1_0.has_message = TRUE
  n1_1.has_message = FALSE
  n1_2.has_message = FALSE
  n1_3.has_message = FALSE
  n1_4.has_message = FALSE
  e1_01.send_a_to_b = FALSE
  e1_01.send_b_to_a = FALSE
  e1_02.send_a_to_b = FALSE
  e1_02.send_b_to_a = FALSE
  e1_03.send_a_to_b = FALSE
  e1_03.send_b_to_a = FALSE
  e1_04.send_a_to_b = FALSE
  e1_04.send_b_to_a = FALSE
  e1_14.send_a_to_b = FALSE
  e1_14.send_b_to_a = FALSE
  e1_23.send_a_to_b = FALSE
  e1_23.send_b_to_a = FALSE

```

Figure 5.8: Toplogy2 terminates before Topology1 with initial node 0.

As *Topology1* has been proved to terminate before *Topology2* with initial node 0, we consider the possibility of *Topology1* terminating before *Topology2* regardless of the initial node chosen for each. By the discussion in subsection 5.4.4, this means replacing  $\phi_i^1$  and  $\phi_i^2$  by the following  $\phi_{i_v}^1$  and  $\phi_{i_v}^2$ , respectively:

$$\begin{aligned}
\phi_{i_v}^1 &\equiv (M_0^1 \wedge \neg M_1^1 \wedge \neg M_2^1 \wedge \neg M_3^1 \wedge \neg M_4^1 \vee \\
&\quad \neg M_0^1 \wedge M_1^1 \wedge \neg M_2^1 \wedge \neg M_3^1 \wedge \neg M_4^1 \vee \\
&\quad \neg M_0^1 \wedge \neg M_1^1 \wedge M_2^1 \wedge \neg M_3^1 \wedge \neg M_4^1 \vee \\
&\quad \neg M_0^1 \wedge \neg M_1^1 \wedge \neg M_2^1 \wedge M_3^1 \wedge \neg M_4^1 \vee \\
&\quad \neg M_0^1 \wedge \neg M_1^1 \wedge \neg M_2^1 \wedge \neg M_3^1 \wedge M_4^1) \\
\phi_{i_v}^2 &\equiv (M_0^2 \wedge \neg M_1^2 \wedge \neg M_2^2 \wedge \neg M_3^2 \wedge \neg M_4^2 \vee
\end{aligned}$$

$$\begin{aligned}
& \neg M_0^2 \wedge M_1^2 \wedge \neg M_2^2 \wedge \neg M_3^2 \wedge \neg M_4^2 \vee \\
& \neg M_0^2 \wedge \neg M_1^2 \wedge M_2^2 \wedge \neg M_3^2 \wedge \neg M_4^2 \vee \\
& \neg M_0^2 \wedge \neg M_1^2 \neg \wedge \neg M_2^2 \wedge M_3^2 \wedge \neg M_4^2 \vee \\
& \neg M_0^2 \wedge \neg M_1^2 \neg \wedge \neg M_2^2 \wedge \neg M_3^2 \wedge M_4^2
\end{aligned}$$

So, the proof obligation for *Topology1* always terminating before *Topology2*, for a random choice of initial nodes, is to check the validity of:

$$(\phi_m^1 \wedge \phi_{i_v}^1) \wedge (\phi_m^2 \wedge \phi_{i_v}^2) \Rightarrow$$

$$\mathbf{F}((\neg M_0^1 \wedge \neg M_1^1 \wedge \neg M_2^1 \wedge \neg M_3^1 \wedge \neg M_4^1) \wedge (M_0^2 \vee M_1^2 \vee M_2^2 \vee M_3^2 \vee M_4^2))$$

where we substitute the  $\phi_{i_v}^1$  and  $\phi_{i_v}^2$  given above. The result of executing the proof in NuSMV is *True*, i.e., *Topology1* terminates before *Topology2* regardless of the starting nodes chosen. The code is shown in Appendix B Section B.3. This result is shown in Figure 5.9.

```

*** This is NuSMV 2.5.4 (compiled on Fri Nov 23 21:36:06 UTC 2012)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>

*** Copyright (c) 2010, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification F (t1.is_terminated & !t2.is_terminated) is true
eduroam-visitor-pt1-175-190:~ Raed$ █

```

Figure 5.9: Topology1 terminates before Topology2 always.

## 5.7 Conclusions

We have provided a specification of network flooding in propositional linear temporal logic suitable for proving termination properties. The specification can cater for any class of graph topologies for a given size of network; it does not cater for networks of arbitrary size, however. A temporal-logic specification of flooding for networks of arbitrary size would need to use first-order temporal logic. Although

first-order temporal logic can *specify* problems of unlimited size - for example, the specification of a transactional system over an unbounded number of data items given in [54] - there are practical and theoretical obstacles to formal verification in such logics. Even with the specifications here, and the use of one of the most powerful model checkers available, NuSMV, proofs will only be possible in practice for fairly small network sizes. Nevertheless, experimentation with network topologies on a small scale can provide insight into the design of networks on a larger scale. The intended use of the approach here is to facilitate the design of network hardware and software by experimentation with different topologies and also different code/algorithms at the nodes. The flooding problem gives an example of a very basic algorithm at a network node - on receipt of a message, a node sends on the message to all the neighbours from which it did not receive the message. In the same way as network topologies can easily be changed by changing the topological constraints, so too can the code/algorithm at nodes be changed by supplying new, possibly more sophisticated, message-processing constraints, which can then be verified.



# Chapter 6

## Asynchronous Network Flooding

### 6.1 Introduction

In the synchronous model presented in the previous chapter the sending and receipt of a message occurred in the same unit of time that was used, that is, in the same ‘round’. Here, we introduce the possibility that the sending and receipt of a message may occur during different time units. If during some unit in time a message is sent but is not received in that unit in time, we say that the message is in ‘transit’. The two main primitives will be:

- $T_{g,h}$ : a message from node  $g$  to node  $h$  is in transit
- $R_{g,h}$ : a message from node  $g$  to node  $h$  is received

The sending of a message represented by the proposition  $S_{i,j}$  in the previous chapter will correspond to a sent message either being in transit or already received. Allowing for the possibility of a sent message not being received immediately gives our model an element of asynchrony. The asynchronous flooding model will be fashioned by the kind of properties we are interested in proving. As with the synchronous case, one of our aims is to be able to compare time to termination for different topologies. However, it is not interesting to compare the termination time between two (non-deterministic) asynchronous networks as it is fairly easy to see that flooding in any such non-trivial network with cycles can continue for an arbitrarily long time. Thus, flooding in any one of two such networks could terminate before the other by suitably delaying termination.

We are interested in comparing synchronous and asynchronous networks to see if asynchronicity can result in earlier termination. This means being able to relate ‘rounds’ in the synchronous case to the asynchronous case. We describe a formal model of asynchrony in Section 6.2 along with a generalized notion of ‘round’ for the asynchronous case. This will be presented as a model in linear temporal

logic in Section 6.3. In Section 6.4, we will give proof obligations to show that an asynchronous model can terminate earlier than a synchronous model, and use these proof obligations to verify examples in NuSMV. Section 6.5 discusses the achievements of this chapter.

## 6.2 An Asynchronous Network Flooding Model

In order to define ‘rounds’ for an asynchronous model, it is convenient to define asynchronous flooding formally as a state transition system  $(\mathcal{S}, \rightarrow)$ , where  $\mathcal{S}$  are the states and  $\rightarrow$  is the transition relation, so that we can clearly identify transitions that we want to exclude as rounds. The states in the state transition system will be sets of actions which we define beforehand.

**Definition 6.2.1** *Let  $N$  be a finite set of nodes of a network. An action on  $N$  is one of the following actions for nodes  $g, h \in N$ :*

- A1 an action  $T_{g,h}$  indicating a message is in transit from node  $g$  to node  $h$ ;*
- A2 an action  $R_{g,h}$  indicating a message from node  $g$  to node  $h$  is received;*
- A3 an action  $M_g$  indicating a message has been received at node  $g$  from some other node, or  $g$  holds a message as the initial node.*

**Definition 6.2.2** *Suppose that  $(N, E)$  is the graph of a network with a set of nodes  $N$  and a set of undirected edges  $E$ , and let  $g_0 \in N$  be a fixed initial node. A state  $s \in \mathcal{S}$  is either the set  $s_0 = \{M_{g_0}\}$ , called the initial state, or a finite set of actions on  $N$  satisfying the following conditions:*

- S1 for all  $g, h \in N$ , actions  $T_{g,h}$  or  $R_{g,h}$  can only belong to  $s$  if there is an edge between  $g$  and  $h$  in  $E$ ;*
- S2 for all  $g \in N$  and states  $s$  not equal to the initial state,  $M_g$  belongs to  $s$  iff  $R_{h,g}$  belongs to  $s$  for some  $h \in N$ .*

To define the state transition system, we also need to define the transition relation.

**Definition 6.2.3** *Given states  $s, s' \in \mathcal{S}$ , there is a transition  $s \rightarrow s'$  iff, for all  $g, h \in N$ , the following conditions are satisfied:*

- TS1 if  $M_g \in s$  and  $R_{h,g}, T_{g,h} \notin s$ , then either  $T_{g,h} \in s'$  and  $R_{g,h} \notin s'$  or  $R_{g,h} \in s'$  and  $T_{g,h} \notin s'$ ;*
- TS2 if  $M_g, T_{g,h} \in s$ , and  $R_{h,g} \notin s$ , then  $T_{g,h}, R_{g,h} \in s'$ ;*

*TS3* if  $R_{h,g} \in s$  and  $T_{g,h} \notin s$ , then  $T_{g,h}, R_{g,h} \notin s'$ ;

*TS4* if  $T_{g,h} \in s$ , then  $R_{g,h} \in s'$ , and  $T_{g,h} \in s'$  if and only if  $M_g \in s$  and  $R_{h,g} \notin s$ ;

*TS5* if  $M_g, T_{g,h} \notin s$ , then  $T_{g,h}, R_{g,h} \notin s'$ .

Here,  $s$  was the last state of the network before the current state of the network,  $s'$ . Condition *TS1* states that if there was a message at node  $g$  which was not received from node  $h$ , and there was no message in transit from  $g$  to  $h$ , then  $g$  sends a message to  $h$  which may be in transit to  $h$  or may have been received by  $h$ . Condition *TS2* states that if there was a message at node  $g$  which was not received from node  $h$  and there was already a message in transit from  $g$  to  $h$ , then the message that was in transit from  $g$  to  $h$  has been received and another message from  $g$  to  $h$  is in transit. Condition *TS3* states that if node  $h$  has just sent a message to node  $g$  and there was no message in transit from  $g$  to  $h$ , then no message sent from  $g$  to  $h$  is either in transit or has been received by  $h$ . Condition *TS4* states that, no matter what the state is of other nodes and messages in the network, a message that was in transit from node  $g$  to node  $h$  in one state of the network is received by  $h$  in the next state of the network, and also that there is another message in transit from  $g$  to  $h$  if, and only if,  $g$  had a message in  $s$  and did not receive a message from  $h$  in the last state. Condition *TS5* states that if node  $g$  did not have a message and there was not already a message in transit from  $g$  to a node  $h$ , then there is no message from  $g$  to  $h$ , either in transit or received, in the current state.

We define executions or ‘runs’ of the asynchronous network in terms of the state transition system.

**Definition 6.2.4** Let  $(N, E)$  be the graph of a network with set of nodes  $N$  and set of edges  $E$ , and let  $g_0$  be the initial node. Let  $(\mathcal{S}, \rightarrow)$  be the asynchronous flooding state transition system for this network and initial node, as defined above. A run of the flooding algorithm, with respect to  $G = (N, E, g_0)$ , is a sequence of states of  $\mathcal{S}$

$$s_0, s_1, \dots, s_i, \dots$$

such that:

$$R1 \quad s_0 = \{g_0\},$$

$$R2 \quad s_{i-1} \rightarrow s_i \text{ for all } i \geq 1.$$

This definition of runs allows for states where no node holds a message, as all messages are in transit (see the example below). In the synchronous model, at least one node in every state of the network, before termination, holds a message.

Our extension of the notion of ‘rounds’ to asynchronous flooding is to states in the state transition system where some node holds a message. A run in which all states, before termination, have nodes with messages, will be said to be in ‘round form’.

**Definition 6.2.5** Let  $G = (N, E, g_0)$  be as per the definition of runs above, and let  $\mathcal{R}_G$  denote the corresponding set of all runs. A run  $r \in \mathcal{R}_G$

$$r = s_0, s_1, \dots, s_i, \dots$$

is in round form iff, for all  $i \geq 0$ ,

$$M_g \in s_i \text{ for some } g \in N.$$

**Example 6.2.6** Consider the following runs:

$$\begin{aligned} r_1 &= \{M_0\}, \{R_{0,1}, R_{0,2}, M_1, M_2\}, \{R_{1,2}, R_{2,1}, M_1, M_2\}, \{R_{1,0}, R_{2,0}, M_0\}, \{\} \\ r_2 &= \{M_0\}, \{T_{0,1}, T_{0,2}\}, \{R_{0,1}, R_{0,2}, M_1, M_2\}, \{R_{1,2}, R_{2,1}, M_1, M_2\}, \\ &\quad \{R_{1,0}, R_{2,0}, M_0\}, \{\} \\ r_3 &= \{M_0\}, \{R_{0,1}, T_{0,2}, M_1\}, \{R_{1,2}, R_{0,2}, M_2\}, \{\} \end{aligned}$$

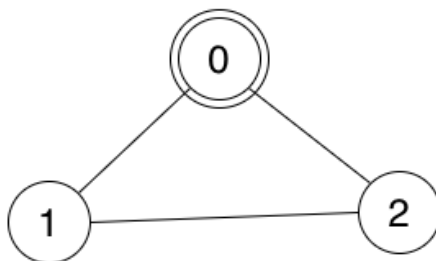


Figure 6.1: Example 2.6

Here,  $r_1$  is in round form and corresponds to the synchronous execution of  $G$ . The run  $r_2$  is not in round form as the second state,  $\{T_{0,1}, T_{0,2}\}$ , has no  $M_g$  action. Note that  $r_3$  is in round form as every state, prior to termination, has some  $M_g$  action. However, it is different to  $r_1$  and does not correspond to a synchronous run. Interestingly, the delay in transit of the message from node 0 to node 2 in  $r_3$  results in earlier termination for  $r_3$ , i.e., termination is achieved in fewer rounds.

When choosing a suitable measure of the ‘time taken’ for a run to terminate, the duration of non-round states, such as  $\{T_{0,1}, T_{0,2}\}$  of  $r_2$  above, has no meaning in terms of numbers of ‘observable’ events in network flooding, i.e., the events  $M_g$  of messages arriving at nodes. In fact, as we show in the theorem below, we can delete all non-round states from a run in  $\mathcal{R}_G$  and the resulting sub-sequence of states will still be a run in  $\mathcal{R}_G$  and will be in round form. Therefore, when considering the runs in  $\mathcal{R}_G$  which terminate the quickest, we can confine our attention to runs in round form, and the measure of the time taken to termination is, as in the synchronous case, the number of rounds to termination.

**Theorem 6.2.7** *Let  $G = (N, E, g_0)$  be a network graph  $(N, E)$  along with an initial node  $g_0 \in N$  as above. Suppose that  $r \in \mathcal{R}_G$  is a run and that  $r_{RF}$  is the sub-sequence of states of  $r$  which are rounds. Then,  $r_{RF}$  is a run in  $\mathcal{R}_G$  (and is in round form).*

*Proof* It suffices to show that if we remove the first state that is not a round, then the resulting sequence of states is still a run. The theorem then follows by repeatedly removing the first occurrences of non-round states in a similar manner until no non-round states remain and we are left with a run  $r_{RF}$  in round form. Let

$$r = s_0, s_1, \dots, s_{i-1}, s_i, s_{i+1}, \dots$$

where  $i \geq 1$ ,  $s_0, \dots, s_{i-1}$  are rounds and  $s_i$  is not a round. Consider the subsequent  $r^{-i}$  of  $r$  given by

$$r^{-i} = s_0, \dots, s_{i-1}, s_{i+1}, \dots \quad (6.1)$$

We need to show that  $r^{-i}$  is a run. Let  $(\mathcal{S}, \rightarrow)$  be the state transition system, as in Definitions 6.2.1 - 6.2.3, generating runs in  $\mathcal{R}_G$ . To show that  $r^{-i}$  is a run in  $\mathcal{R}_G$ , we need to show that  $r^{-i}$  can be generated by  $(\mathcal{S}, \rightarrow)$ . As  $(\mathcal{S}, \rightarrow)$  generates a state  $s_{j+1}$  from the previous state  $s_j$  alone, by the conditions *TS1-TS5* of Definition 6.2.3, it is clear that in both  $r$  and  $r^{-i}$ :  $s_1$  are generated from  $s_0$ ,  $\dots$ ,  $s_{i-1}$  is generated from  $s_{i-2}$ ,  $s_{i+2}$  is generated from  $s_{i+1}$ ,  $\dots$ . Therefore, for  $r^{-i}$  to be a run in  $\mathcal{R}_G$ , it only remains to show that  $s_{i+1}$  can be generated from  $s_{i-1}$ . Now, as  $s_i$  is not a round,  $s_i$  has no  $M_g$  and thus no  $R_{g,h}$  actions. So,  $s_i$  consists entirely of  $T_{g,h}$  actions. Let

$$s_i = \{T_{g_1, h_1}, \dots, T_{g_k, h_k}\} \quad (6.2)$$

where  $k \geq 1$ . As  $r$  is a run, we know that  $s_{i+1}$  is generated from  $s_i$ . The only condition for  $(\mathcal{S}, \rightarrow)$  that generates a state from a state with only  $T_{g,h}$  actions is *TS4*, which produces the corresponding set of  $R_{g,h}$  actions. Thus, from (6.2),

$$s_{i+1} = \{R_{g_1, h_1}, \dots, R_{g_k, h_k}\} \quad (6.3)$$

Also, as  $r$  is a run,  $s_{i-1}$  generates  $s_i$ . The only condition for  $(\mathcal{S}, \rightarrow)$  that generates  $T_{g,h}$  but does not generate  $R_{g,h}$  is  $TS1$ . Thus,

$$\{M_{g_1}, \dots, M_{g_k}\} \subseteq s_{i-1} \text{ and } \{R_{h_1, g_1}, \dots, R_{h_k, g_k}, T_{g_1, h_1}, \dots, T_{g_k, h_k}\} \cap s_{i-1} = \emptyset$$

However, condition  $TS1$  allows  $R_{g,h}$  to be generated instead of the corresponding  $T_{g,h}$ . Thus, if we generate  $R_{g_1, h_1}, \dots, R_{g_k, h_k}$  instead of  $T_{g_1, h_1}, \dots, T_{g_k, h_k}$  of  $s_i$  in (6.2), we get exactly the state  $s_{i+1}$  of (6.3). Hence,  $s_{i+1}$  can be generated from  $s_{i-1}$ , and so the  $r^{-i}$  of (6.1) is a run in  $\mathcal{R}_G$ . ■

The importance of this theorem when comparing asynchronous and synchronous systems is explained in Section 6.4 below.

### 6.3 Temporal Model

Given a network graph and initial node  $G = (N, E, g_0)$ , the states of the linear temporal logic structure that models  $G$  will be exactly the states of the state transition system  $(\mathcal{S}, \rightarrow)$  of  $G$ , as given in Section 6.2 above.

#### 6.3.1 Propositions

There is a proposition corresponding to each action in  $(\mathcal{S}, \rightarrow)$ . Thus, the sets of propositions are:

- (i) *message in transit from  $g$  to  $h$  propositions:*  $\{T_{g,h} \mid \{g, h\} \text{ is an edge in } E\}$ ;
- (ii) *message received from  $g$  to  $h$  propositions:*  $\{R_{g,h} \mid \{g, h\} \text{ is an edge in } E\}$ ;
- (iii) *message received at node  $g$  propositions:*  $\{M_g \mid g \text{ is a node in } N\}$ .

#### 6.3.2 States

In Definition 6.2.2, a state  $s$  in  $(\mathcal{S}, \rightarrow)$  is a finite sets of actions. The corresponding temporal state will have the propositions of the actions in  $s$  returned as *True*. However, condition  $S2$  of Definition 6.2.2 which requires that a  $M_g$  action belong to a (non-initial) state iff a  $R_{h,g}$  action that also belongs to some node  $h$ , needs the following additional temporal constraint:

$$\phi_{S2} \equiv \mathbf{XG} \bigwedge_{g \in N} (M_g \Leftrightarrow \bigvee_{\substack{h \in N, \\ h \neq g}} R_{h,g})$$

### 6.3.3 Run constraints

We lift the run constraints directly from the transition relation  $\rightarrow$  conditions *TS1-TS5* of Definition 6.2.3. Condition *TS1* requires that, for all nodes  $g$  and  $h$ , if an  $M_g$  action belongs to the previous state and  $R_{h,g}$  and  $T_{g,h}$  do not, then either  $T_{g,h}$  belongs to the current state and  $R_{g,h}$  does not, or  $R_{g,h}$  belongs to the current state and  $T_{g,h}$  does not. The temporal constraint in terms of the truth values of the propositions is:

$$\phi_{TS1} \equiv \mathbf{XG} \bigwedge_{\substack{g,h \in N, \\ g \neq h}} ((\mathbf{Y}(M_g \wedge \neg R_{h,g} \wedge \neg T_{g,h})) \Rightarrow ((T_{g,h} \wedge \neg R_{g,h}) \vee (R_{g,h} \wedge \neg T_{g,h})))$$

The leftmost  $\mathbf{X}$  is needed as the *TS1-TS5* conditions are from the point of view of a state with a previous state, so we start after the initial state. Condition *TS2* requires that, for all nodes  $g$  and  $h$ , if the  $M_g$  and  $T_{g,h}$  actions belong to the previous state and  $R_{h,g}$  does not, then both  $T_{g,h}$  and  $R_{g,h}$  belong to the current state. The temporal constraint is:

$$\phi_{TS2} \equiv \mathbf{XG} \bigwedge_{\substack{g,h \in N, \\ g \neq h}} ((\mathbf{Y}(M_g \wedge T_{g,h} \wedge \neg R_{h,g})) \Rightarrow (T_{g,h} \wedge R_{g,h}))$$

Condition *TS3* requires that, for all nodes  $g$  and  $h$ , if a  $R_{h,g}$  action belongs to the previous state and  $T_{g,h}$  does not, then neither  $T_{g,h}$  nor  $R_{g,h}$  belong to the current state. The temporal constraint is:

$$\phi_{TS3} \equiv \mathbf{XG} \bigwedge_{\substack{g,h \in N, \\ g \neq h}} ((\mathbf{Y}(R_{h,g} \wedge \neg T_{g,h})) \Rightarrow (\neg T_{g,h} \wedge \neg R_{g,h}))$$

Condition *TS4* requires that, for all nodes  $g$  and  $h$ , if a  $T_{g,h}$  action belongs to the previous state, then  $R_{g,h}$  belongs to the current state and  $T_{g,h}$  is in the current state iff  $M_g$  was in the previous state and  $R_{h,g}$  was not. The temporal constraint is:

$$\phi_{TS4} \equiv \mathbf{XG} \bigwedge_{\substack{g,h \in N, \\ g \neq h}} ((\mathbf{Y}T_{g,h}) \Rightarrow R_{g,h} \wedge (T_{g,h} \Leftrightarrow \mathbf{Y}(M_g \wedge \neg R_{h,g})))$$

Condition *TS5* requires that, for all nodes  $g$  and  $h$ , if neither a  $M_g$  nor a  $T_{g,h}$  action belongs to the previous state, then neither  $T_{g,h}$  nor  $R_{g,h}$  belongs to the current state. The temporal constraint is:

$$\phi_{TS5} \equiv \mathbf{XG} \bigwedge_{\substack{g,h \in N, \\ g \neq h}} ((\mathbf{Y}(\neg M_g \wedge \neg T_{g,h})) \Rightarrow (\neg T_{g,h} \wedge \neg R_{g,h}))$$

The conditions  $TS1$ - $TS5$  describe synchronous runs if no  $T_{g,h}$  action appears in any state in  $(\mathcal{S}, \rightarrow)$ . This corresponds to replacing all  $T_{g,h}$  propositions in  $\phi_{TS1}$ - $\phi_{TS5}$  by *False*. In this case, if we also substitute  $R_{g,h}$  propositions by  $S_{g,h}$  propositions in the conjunction  $\phi_{s2} \wedge \phi_{TS1} \wedge \phi_{TS3} \wedge \phi_{TS3}$ , we get a temporal logic formula equivalent to the constraint  $\phi_m$  for received messages in the synchronous flooding described in the previous chapter.

We also need to add a temporal constraint to the initial state of  $(\mathcal{S}, \rightarrow)$  to be the set  $\{M_{g_0}\}$ . This means the proposition  $M_{g_0}$  is *True* and all other propositions are *False*. The initial temporal state constraint is:

$$\phi_{init} \equiv (M_{g_0} \wedge \bigwedge_{\substack{g \in N, \\ g \neq g_0}} \neg M_g) \wedge \left( \bigwedge_{\substack{g, h \in N, \\ g \neq h}} (\neg T_{g,h} \wedge \neg R_{g,h}) \right)$$

### 6.3.4 Termination

To specify the termination of a run, it is initially tempting to proceed as in the synchronous case and specify that there is a state in which no node holds a message. However, in the asynchronous case it is possible to have many states where no node holds a message because messages are in transit; clearly, however, if this is the case then the run has not yet terminated. Thus, we need to specify a state in which no node holds a message and no messages are in transit. The temporal constraint for termination is:

$$\phi_{term} \equiv \mathbf{F} \bigwedge_{\substack{g, h \in N, \\ g \neq h}} (\neg M_g \wedge \neg T_{g,h})$$

### 6.3.5 Rounds

In Section 6.4 below we will compare the time to termination of synchronous and asynchronous runs of network flooding. As discussed in Section 6.2, we do so by counting the rounds in runs of both types. For this, we need all the states of the asynchronous runs to be rounds, i.e., some node has to receive a message in every state before the run terminates. This can be expressed by the constraint that no state can have a  $T_{g,h}$  action and no  $R_{g,h}$  actions. This does not affect terminated states, which have no  $T_{g,h}$  and no  $R_{g,h}$  actions. The temporal constraint is:

$$\phi_{round} \equiv \mathbf{G} \neg \left( \left( \bigvee_{\substack{g, h \in N, \\ g \neq h}} T_{g,h} \right) \wedge \left( \bigwedge_{\substack{g, h \in N, \\ g \neq h}} \neg R_{g,h} \right) \right)$$



## 6.4 Comparing Asynchronous and Synchronous Termination

In the previous chapter, we compared the number of rounds taken to termination in two network topologies by superimposing successive states of rounds in one topology over the successive states of rounds in the other. In this section, we will compare the number of rounds to termination of flooding between a terminating asynchronous network and a synchronous network with the same topology to see if the asynchronous network can terminate in fewer rounds. We will assume a single fixed set of nodes and graph topology for both cases and the same initial node. From the previous chapter, the states in the synchronous network in successive rounds are the model of the temporal logic formula:

$$\phi_m \wedge \phi_i, \quad (6.4)$$

where  $\phi_m$  are the message-received constraints and  $\phi_i$  are the initial conditions. In this chapter, the succession of states in the asynchronous case, here called runs, are the models of the temporal logic formula:

$$\phi_{S2} \wedge \phi_{TS1} \wedge \phi_{TS2} \wedge \phi_{TS3} \wedge \phi_{TS4} \wedge \phi_{TS5} \wedge \phi_{init}. \quad (6.5)$$

To prove that there are runs in the asynchronous case that can terminate in fewer rounds than the synchronous case, we might, tentatively, consider checking for the validity of the following temporal formula:

$$\neg(\phi_{S2} \wedge \phi_{TS1} \wedge \phi_{TS2} \wedge \phi_{TS3} \wedge \phi_{TS4} \wedge \phi_{TS5} \wedge \phi_{init} \wedge \phi_m \wedge \phi_i) \quad (6.6)$$

$$\Rightarrow \mathbf{F}\left(\bigwedge_{\substack{g,h \in N, \\ g \neq h}} (\neg M_g \wedge \neg T_{g,h}) \wedge \left(\bigvee_{g \in N} M_g^s\right)\right) \quad (6.7)$$

Line (6.6) represents the states of the superimposed asynchronous and synchronous networks and line (6.7) asserts that at some point in time there will be no further asynchronous actions (i.e., the asynchronous case has terminated) but there will be some synchronous actions (i.e., the synchronous case has not terminated). The whole formula spread over lines (6.6) and (6.7) has an outer negation. It is quite literally asserting that, for all asynchronous and synchronous runs, it is not true that the asynchronous case terminates before the synchronous. For there to be an asynchronous run that terminates before the synchronous, the test for the validity of formula (6.6),(6.7) should return *False*. As we have used mostly different propositional variables for asynchronous systems in (6.5) in this chapter to those

for synchronous systems in (6.4) in the previous chapter, the only variables we need to relabel are the  $M_g$  variables, which we relabel only in the synchronous case to have a superscript  $s$ .

However, there is a problem with the formula (6.6),(6.7). The behaviour of the asynchronous system in line (6.5) is for runs which may have states that are not rounds. It is possible that an asynchronous run may terminate in fewer rounds than a synchronous run, but this will not show up in the temporal logic formula (6.6),(6.7) as successive *states* of the asynchronous runs are in lock-step with *rounds* of the synchronous runs, and the asynchronous run may have states that are not rounds. Thus, an asynchronous run may take more states to terminate than the synchronous run takes rounds (this will show up in (6.6),(6.7)) but the asynchronous run may actually take fewer rounds of its own (this will not show up in (6.6),(6.7)). We could restrict the asynchronous runs to those in round form by adding the extra conjunct  $\phi_{round}$  to the asynchronous behaviour, yielding the proof obligation:

$$\neg(\phi_{S2} \wedge \phi_{TS1} \wedge \phi_{TS2} \wedge \phi_{TS3} \wedge \phi_{TS4} \wedge \phi_{TS5} \wedge \phi_{round} \wedge \phi_{init} \wedge \phi_m \wedge \phi_i) \quad (6.8)$$

$$\Rightarrow \mathbf{F}\left(\bigwedge_{\substack{g,h \in N, \\ g \neq h}} (\neg M_g \wedge \neg T_{g,h}) \wedge \left(\bigvee_{g \in N} M_g^s\right)\right) \quad (6.9)$$

However,  $\phi_{round}$  does not eliminate non-round states in an asynchronous run, but eliminates the *whole* run if there is a state that is not a round. It could be the case that a run that has states that are not rounds is nevertheless the one that terminates in the fewest rounds. Fortunately, this is not a problem if we invoke Theorem 6.2.7. Theorem 6.2.7 states that if we delete all non-round states in a run, the remaining sub-sequence of states is still a run. So, even though  $\phi_{round}$  may eliminate the run which has non-round states but terminates in the fewest number of rounds, Theorem 6.2.7 guarantees that there will be another run identical to the eliminated run, but without its non-round states. Thus, formula (6.8),(6.9) is the required proof obligation that demonstrates that an asynchronous run can(not) terminate before the synchronous run.

### 6.4.1 Worked Examples

In 6.4.1.1, we give an example of a network where asynchronous flooding can terminate before synchronous flooding, and in 6.4.1.1 6.4.1.2 we give an example of a network where it cannot. To check if a property is met, after creating the system model and its properties we encode it into the model checker NuSMV to verify if the property is satisfied. The flooding block diagram representing this is

shown below in Figure 6.2.

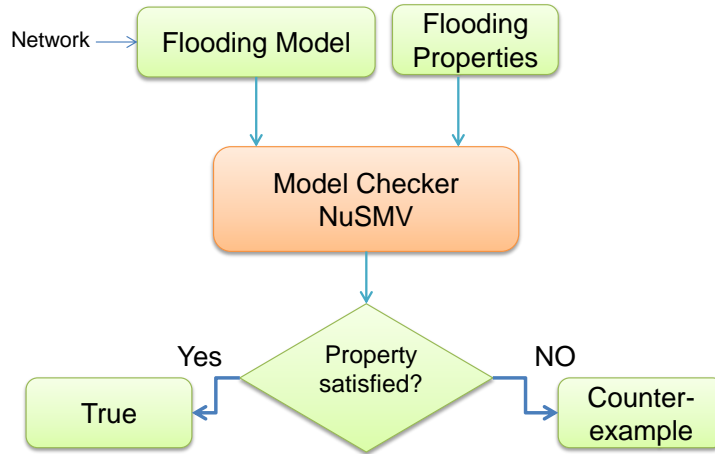


Figure 6.2: Synchronous and Asynchronous flooding model checking block diagram.

#### 6.4.1.1 Asynchronous Can Terminate Before Synchronous

We will verify formally that the network in Example 6.2.6, Figure 5.1, has an asynchronous run which terminates in fewer rounds than the synchronous run with initial nodes 0 using NuSMV. We have the following propositions for the asynchronous and synchronous cases, respectively:

$$\begin{aligned}
 \textit{Asynchronous} : & \quad T_{0,1}, T_{1,0}, T_{0,2}, T_{2,0}, T_{1,2}, T_{2,1}, \\
 & \quad R_{0,1}, R_{1,0}, R_{0,2}, R_{2,0}, R_{1,2}, R_{2,1}, \\
 & \quad M_0, M_1, M_2 \\
 \textit{Synchronous} : & \quad S_{0,1}, S_{1,0}, S_{0,2}, S_{2,0}, S_{1,2}, S_{2,1}, \\
 & \quad M_0^s, M_1^s, M_2^s
 \end{aligned}$$

For the asynchronous case, instantiating the definitions of  $\phi_{S2}$  of Subsection 6.3.2,  $\phi_{TS1}$ - $\phi_{TS5}$  and  $\phi_{init}$  of Subsection 6.3.3, and  $\phi_{round}$  of Subsection 6.3.5, we have:

$$\begin{aligned}\phi_{S2} \equiv \mathbf{XG} \quad & ((M_0 \Leftrightarrow R_{1,0} \vee R_{2,0}) \wedge \\ & (M_1 \Leftrightarrow R_{0,1} \vee R_{2,1}) \wedge \\ & (M_2 \Leftrightarrow S_{0,2} \vee S_{1,2}))\end{aligned}$$

$$\begin{aligned}\phi_{TS1} \equiv \mathbf{XG} \quad & (((\mathbf{Y}(M_0 \wedge \neg R_{1,0} \wedge \neg T_{0,1})) \Rightarrow ((T_{0,1} \wedge \neg R_{0,1}) \vee (R_{0,1} \wedge \neg T_{0,1}))) \wedge \\ & ((\mathbf{Y}(M_1 \wedge \neg R_{0,1} \wedge \neg T_{1,0})) \Rightarrow ((T_{1,0} \wedge \neg R_{1,0}) \vee (R_{1,0} \wedge \neg T_{1,0}))) \wedge \\ & ((\mathbf{Y}(M_0 \wedge \neg R_{2,0} \wedge \neg T_{0,2})) \Rightarrow ((T_{0,2} \wedge \neg R_{0,2}) \vee (R_{0,2} \wedge \neg T_{0,2}))) \wedge \\ & ((\mathbf{Y}(M_2 \wedge \neg R_{0,2} \wedge \neg T_{2,0})) \Rightarrow ((T_{2,0} \wedge \neg R_{2,0}) \vee (R_{2,0} \wedge \neg T_{2,0}))) \wedge \\ & ((\mathbf{Y}(M_1 \wedge \neg R_{2,1} \wedge \neg T_{1,2})) \Rightarrow ((T_{1,2} \wedge \neg R_{1,2}) \vee (R_{1,2} \wedge \neg T_{1,2}))) \wedge \\ & ((\mathbf{Y}(M_2 \wedge \neg R_{1,2} \wedge \neg T_{2,1})) \Rightarrow ((T_{2,1} \wedge \neg R_{2,1}) \vee (R_{2,1} \wedge \neg T_{2,1}))))\end{aligned}$$

$$\begin{aligned}\phi_{TS2} \equiv \mathbf{XG} \quad & (((\mathbf{Y}(M_0 \wedge \neg R_{1,0} \wedge T_{0,1})) \Rightarrow (T_{0,1} \wedge R_{0,1})) \wedge \\ & ((\mathbf{Y}(M_1 \wedge \neg R_{0,1} \wedge T_{1,0})) \Rightarrow (T_{1,0} \wedge R_{1,0})) \wedge \\ & ((\mathbf{Y}(M_0 \wedge \neg R_{2,0} \wedge T_{0,2})) \Rightarrow (T_{0,2} \wedge R_{0,2})) \wedge \\ & ((\mathbf{Y}(M_2 \wedge \neg R_{0,2} \wedge T_{2,0})) \Rightarrow (T_{2,0} \wedge R_{2,0})) \wedge \\ & ((\mathbf{Y}(M_1 \wedge \neg R_{2,1} \wedge T_{1,2})) \Rightarrow (T_{1,2} \wedge R_{1,2})) \wedge \\ & ((\mathbf{Y}(M_2 \wedge \neg R_{1,2} \wedge T_{2,1})) \Rightarrow (T_{2,1} \wedge R_{2,1})))\end{aligned}$$

$$\begin{aligned}\phi_{TS3} \equiv \mathbf{XG} \quad & (((\mathbf{Y}(R_{1,0} \wedge \neg T_{0,1})) \Rightarrow (\neg T_{0,1} \wedge \neg R_{0,1})) \wedge \\ & ((\mathbf{Y}(R_{0,1} \wedge \neg T_{1,0})) \Rightarrow (\neg T_{1,0} \wedge \neg R_{1,0})) \wedge \\ & ((\mathbf{Y}(R_{2,0} \wedge \neg T_{0,2})) \Rightarrow (\neg T_{0,2} \wedge \neg R_{0,2})) \wedge \\ & ((\mathbf{Y}(R_{0,2} \wedge \neg T_{2,0})) \Rightarrow (\neg T_{2,0} \wedge \neg R_{2,0})) \wedge \\ & ((\mathbf{Y}(R_{2,1} \wedge \neg T_{1,2})) \Rightarrow (\neg T_{1,2} \wedge \neg R_{1,2})) \wedge \\ & ((\mathbf{Y}(R_{1,2} \wedge \neg T_{2,1})) \Rightarrow (\neg T_{2,1} \wedge \neg R_{2,1})))\end{aligned}$$

$$\begin{aligned}\phi_{TS4} \equiv \mathbf{XG} \quad & (((\mathbf{Y}T_{0,1}) \Rightarrow R_{0,1}) \wedge \\ & ((\mathbf{Y}T_{1,0}) \Rightarrow R_{1,0}) \wedge \\ & ((\mathbf{Y}T_{0,2}) \Rightarrow R_{0,2}) \wedge \\ & ((\mathbf{Y}T_{2,0}) \Rightarrow R_{2,0}) \wedge \\ & ((\mathbf{Y}T_{1,2}) \Rightarrow R_{1,2}) \wedge \\ & ((\mathbf{Y}T_{2,1}) \Rightarrow R_{2,1}))\end{aligned}$$

$$\begin{aligned}\phi_{TS5} \equiv \mathbf{XG} \quad & (((\mathbf{Y}(\neg M_0 \wedge \neg T_{0,1})) \Rightarrow (\neg T_{0,1} \wedge \neg R_{0,1})) \wedge \\ & ((\mathbf{Y}(\neg M_1 \wedge \neg T_{1,0})) \Rightarrow (\neg T_{1,0} \wedge \neg R_{1,0})) \wedge \\ & ((\mathbf{Y}(\neg M_0 \wedge \neg T_{0,2})) \Rightarrow (\neg T_{0,2} \wedge \neg R_{0,2})) \wedge \\ & ((\mathbf{Y}(\neg M_2 \wedge \neg T_{2,0})) \Rightarrow (\neg T_{2,0} \wedge \neg R_{2,0})) \wedge \\ & ((\mathbf{Y}(\neg M_1 \wedge \neg T_{1,2})) \Rightarrow (\neg T_{1,2} \wedge \neg R_{1,2})) \wedge \\ & ((\mathbf{Y}(\neg M_2 \wedge \neg T_{2,1})) \Rightarrow (\neg T_{2,1} \wedge \neg R_{2,1})))\end{aligned}$$

$$\begin{aligned}
\phi_{round} &\equiv \mathbf{G} \neg((T_{0,1} \vee T_{1,0} \vee T_{0,2} \vee T_{2,0} \vee T_{1,2} \vee T_{2,1}) \wedge \\
&\quad (\neg R_{0,1} \wedge \neg R_{1,0} \wedge \neg R_{0,2} \wedge \neg R_{2,0} \wedge \neg R_{1,2} \wedge \neg R_{2,1})) \\
\phi_{init} &\equiv (M_0 \wedge \neg M_1 \wedge \neg M_2) \wedge \\
&\quad (\neg T_{0,1} \wedge \neg R_{0,1} \wedge \neg T_{1,0} \wedge \neg R_{1,0} \wedge \\
&\quad \neg T_{0,2} \wedge \neg R_{0,2} \wedge \neg T_{2,0} \wedge \neg R_{2,0} \wedge \\
&\quad \neg T_{1,2} \wedge \neg R_{1,2} \wedge \neg T_{2,1} \wedge \neg R_{2,1})
\end{aligned}$$

For the synchronous case, instantiating the  $\phi_m$  and  $\phi_i$  of the previous chapter yields:

$$\begin{aligned}
\phi_i &\equiv (M_0^s \wedge \neg M_1^s \wedge \neg M_2^s) \wedge \\
&\quad (\neg S_{0,1} \wedge \neg S_{1,0} \wedge \neg S_{0,2} \wedge \neg S_{2,0} \wedge \neg S_{1,2} \wedge \neg S_{2,1}) \\
\phi_m &\equiv (\mathbf{XG} ( (M_0^s \Leftrightarrow S_{1,0} \vee S_{2,0}) \wedge \\
&\quad (M_1^s \Leftrightarrow S_{0,1} \vee S_{2,1}) \wedge \\
&\quad (M_2^s \Leftrightarrow S_{0,2} \vee S_{1,2})) ) \wedge \\
&\quad (\mathbf{XG} ( (S_{0,1} \Leftrightarrow \mathbf{Y}(M_0^s \wedge \neg S_{1,0})) \wedge \\
&\quad (S_{1,0} \Leftrightarrow \mathbf{Y}(M_1^s \wedge \neg S_{0,1})) \wedge \\
&\quad (S_{2,0} \Leftrightarrow \mathbf{Y}(M_2^s \wedge \neg S_{0,2})) \wedge \\
&\quad (S_{0,2} \Leftrightarrow \mathbf{Y}(M_0^s \wedge \neg S_{2,0})) \wedge \\
&\quad (S_{1,2} \Leftrightarrow \mathbf{Y}(M_1^s \wedge \neg S_{2,1})) \wedge \\
&\quad (S_{2,1} \Leftrightarrow \mathbf{Y}(M_2^s \wedge \neg S_{1,2}))) )
\end{aligned}$$

By equation (6.8),(6.9) above, in order to demonstrate that no asynchronous run terminates before the synchronous run, we need to prove the following formula with the substitutions for  $\phi_{S2}$ ,  $\phi_{TS1}$ - $\phi_{TS5}$ ,  $\phi_{round}$ ,  $\phi_{init}$ ,  $\phi_i$  and  $\phi_m$ , as given above:

$$\begin{aligned}
&\neg( \phi_{S2} \wedge \phi_{TS1} \wedge \phi_{TS2} \wedge \phi_{TS3} \wedge \phi_{TS4} \wedge \phi_{TS5} \wedge \phi_{round} \wedge \phi_{init} \wedge \phi_m \wedge \phi_i \\
&\quad \Rightarrow \\
&\quad \mathbf{F} ( (\neg T_{0,1} \wedge \neg T_{1,0} \wedge \neg T_{0,2} \wedge \neg T_{2,0} \wedge \neg T_{1,2} \wedge \neg T_{2,1}) \wedge \\
&\quad (\neg M_0 \wedge \neg M_1 \wedge \neg M_2) \wedge (M_0^s \vee M_1^s \vee M_2^s) )
\end{aligned}$$

This proof has been carried out using NuSMV as shown in code in Appendix C Section C.1 and returns *False*, giving an asynchronous run which terminates before the synchronous run.

#### 6.4.1.2 Asynchronous Cannot Terminate Before Synchronous

We will formally verify the network Figure 5.2 below using NuSMV:

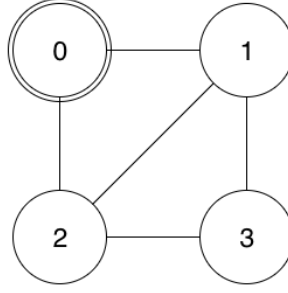


Figure 6.3: Figure of four nodes

This does not have an asynchronous run which terminates in fewer rounds than the synchronous run with initial node 0 for both. We have the following propositions for the asynchronous and synchronous cases, respectively:

$$\begin{aligned}
 \text{Asynchronous : } & T_{0,1}, T_{1,0}, T_{0,2}, T_{2,0}, T_{1,2}, T_{2,1}, T_{1,3}, T_{3,1}, T_{2,3}, T_{3,2} \\
 & R_{0,1}, R_{1,0}, R_{0,2}, R_{2,0}, R_{1,2}, R_{2,1}, R_{1,3}, R_{3,1}, R_{2,3}, R_{3,2}, \\
 & M_0, M_1, M_2, M_3 \\
 \text{Synchronous : } & S_{0,1}, S_{1,0}, S_{0,2}, S_{2,0}, S_{1,2}, S_{2,1}, S_{1,3}, S_{3,1}, S_{2,3}, S_{3,2}, \\
 & M_0^s, M_1^s, M_2^s, M_3^s
 \end{aligned}$$

For the asynchronous case, instantiating the definitions of  $\phi_{S_2}$  of Subsection 6.3.2,  $\phi_{TS1}$ - $\phi_{TS5}$  and  $\phi_{init}$  of Subsection 6.3.3, and  $\phi_{round}$  of Subsection 6.3.5, we have:

$$\begin{aligned}
 \phi_{S_2} \equiv \mathbf{XG} \quad & ((M_0 \Leftrightarrow R_{1,0} \vee R_{2,0}) \wedge \\
 & (M_1 \Leftrightarrow R_{0,1} \vee R_{2,1} \vee R_{3,1}) \wedge \\
 & (M_2 \Leftrightarrow S_{0,2} \vee S_{1,2} \vee S_{3,2}) \wedge \\
 & (M_3 \Leftrightarrow S_{1,3} \vee S_{2,3}))
 \end{aligned}$$

$$\begin{aligned}
 \phi_{TS1} \equiv \mathbf{XG} \quad & (((\mathbf{Y}(M_0 \wedge \neg R_{1,0} \wedge \neg T_{0,1})) \Rightarrow ((T_{0,1} \wedge \neg R_{0,1}) \vee (R_{0,1} \wedge \neg T_{0,1}))) \wedge \\
 & ((\mathbf{Y}(M_1 \wedge \neg R_{0,1} \wedge \neg T_{1,0})) \Rightarrow ((T_{1,0} \wedge \neg R_{1,0}) \vee (R_{1,0} \wedge \neg T_{1,0}))) \wedge \\
 & ((\mathbf{Y}(M_0 \wedge \neg R_{2,0} \wedge \neg T_{0,2})) \Rightarrow ((T_{0,2} \wedge \neg R_{0,2}) \vee (R_{0,2} \wedge \neg T_{0,2}))) \wedge \\
 & ((\mathbf{Y}(M_2 \wedge \neg R_{0,2} \wedge \neg T_{2,0})) \Rightarrow ((T_{2,0} \wedge \neg R_{2,0}) \vee (R_{2,0} \wedge \neg T_{2,0}))) \wedge \\
 & ((\mathbf{Y}(M_1 \wedge \neg R_{2,1} \wedge \neg T_{1,2})) \Rightarrow ((T_{1,2} \wedge \neg R_{1,2}) \vee (R_{1,2} \wedge \neg T_{1,2}))) \wedge \\
 & ((\mathbf{Y}(M_2 \wedge \neg R_{1,2} \wedge \neg T_{2,1})) \Rightarrow ((T_{2,1} \wedge \neg R_{2,1}) \vee (R_{2,1} \wedge \neg T_{2,1}))) \wedge \\
 & ((\mathbf{Y}(M_1 \wedge \neg R_{3,1} \wedge \neg T_{1,3})) \Rightarrow ((T_{1,3} \wedge \neg R_{1,3}) \vee (R_{1,3} \wedge \neg T_{1,3}))) \wedge \\
 & ((\mathbf{Y}(M_3 \wedge \neg R_{1,3} \wedge \neg T_{3,1})) \Rightarrow ((T_{3,1} \wedge \neg R_{3,1}) \vee (R_{3,1} \wedge \neg T_{3,1}))) \wedge \\
 & ((\mathbf{Y}(M_2 \wedge \neg R_{3,2} \wedge \neg T_{2,3})) \Rightarrow ((T_{2,3} \wedge \neg R_{2,3}) \vee (R_{2,3} \wedge \neg T_{2,3}))) \wedge \\
 & ((\mathbf{Y}(M_3 \wedge \neg R_{2,3} \wedge \neg T_{3,2})) \Rightarrow ((T_{3,2} \wedge \neg R_{3,2}) \vee (R_{3,2} \wedge \neg T_{3,2}))))
 \end{aligned}$$

$$\begin{aligned}
\phi_{TS2} \equiv \mathbf{XG} \quad & (((\mathbf{Y}(M_0 \wedge \neg R_{1,0} \wedge T_{0,1})) \Rightarrow (T_{0,1} \wedge R_{0,1})) \wedge \\
& ((\mathbf{Y}(M_1 \wedge \neg R_{0,1} \wedge T_{1,0})) \Rightarrow (T_{1,0} \wedge R_{1,0})) \wedge \\
& ((\mathbf{Y}(M_0 \wedge \neg R_{2,0} \wedge T_{0,2})) \Rightarrow (T_{0,2} \wedge R_{0,2})) \wedge \\
& ((\mathbf{Y}(M_2 \wedge \neg R_{0,2} \wedge T_{2,0})) \Rightarrow (T_{2,0} \wedge R_{2,0})) \wedge \\
& ((\mathbf{Y}(M_1 \wedge \neg R_{2,1} \wedge T_{1,2})) \Rightarrow (T_{1,2} \wedge R_{1,2})) \wedge \\
& ((\mathbf{Y}(M_2 \wedge \neg R_{1,2} \wedge T_{2,1})) \Rightarrow (T_{2,1} \wedge R_{2,1})) \wedge \\
& ((\mathbf{Y}(M_1 \wedge \neg R_{3,1} \wedge T_{1,3})) \Rightarrow (T_{1,3} \wedge R_{1,3})) \wedge \\
& ((\mathbf{Y}(M_3 \wedge \neg R_{1,3} \wedge T_{3,1})) \Rightarrow (T_{3,1} \wedge R_{3,1})) \wedge \\
& ((\mathbf{Y}(M_2 \wedge \neg R_{3,2} \wedge T_{2,3})) \Rightarrow (T_{2,3} \wedge R_{2,3})) \wedge \\
& ((\mathbf{Y}(M_3 \wedge \neg R_{2,3} \wedge T_{3,2})) \Rightarrow (T_{3,2} \wedge R_{3,2})))
\end{aligned}$$

$$\begin{aligned}
\phi_{TS3} \equiv \mathbf{XG} \quad & (((\mathbf{Y}(R_{1,0} \wedge \neg T_{0,1})) \Rightarrow (\neg T_{0,1} \wedge \neg R_{0,1})) \wedge \\
& ((\mathbf{Y}(R_{0,1} \wedge \neg T_{1,0})) \Rightarrow (\neg T_{1,0} \wedge \neg R_{1,0})) \wedge \\
& ((\mathbf{Y}(R_{2,0} \wedge \neg T_{0,2})) \Rightarrow (\neg T_{0,2} \wedge \neg R_{0,2})) \wedge \\
& ((\mathbf{Y}(R_{0,2} \wedge \neg T_{2,0})) \Rightarrow (\neg T_{2,0} \wedge \neg R_{2,0})) \wedge \\
& ((\mathbf{Y}(R_{2,1} \wedge \neg T_{1,2})) \Rightarrow (\neg T_{1,2} \wedge \neg R_{1,2})) \wedge \\
& ((\mathbf{Y}(R_{1,2} \wedge \neg T_{2,1})) \Rightarrow (\neg T_{2,1} \wedge \neg R_{2,1})) \wedge \\
& ((\mathbf{Y}(R_{3,1} \wedge \neg T_{1,3})) \Rightarrow (\neg T_{1,3} \wedge \neg R_{1,3})) \wedge \\
& ((\mathbf{Y}(R_{1,3} \wedge \neg T_{3,1})) \Rightarrow (\neg T_{3,1} \wedge \neg R_{3,1})) \wedge \\
& ((\mathbf{Y}(R_{3,2} \wedge \neg T_{2,3})) \Rightarrow (\neg T_{2,3} \wedge \neg R_{2,3})) \wedge \\
& ((\mathbf{Y}(R_{2,3} \wedge \neg T_{3,2})) \Rightarrow (\neg T_{3,2} \wedge \neg R_{3,2})))
\end{aligned}$$

$$\begin{aligned}
\phi_{TS4} \equiv \mathbf{XG} \quad & (((\mathbf{Y}T_{0,1}) \Rightarrow R_{0,1}) \wedge \\
& ((\mathbf{Y}T_{1,0}) \Rightarrow R_{1,0}) \wedge \\
& ((\mathbf{Y}T_{0,2}) \Rightarrow R_{0,2}) \wedge \\
& ((\mathbf{Y}T_{2,0}) \Rightarrow R_{2,0}) \wedge \\
& ((\mathbf{Y}T_{1,2}) \Rightarrow R_{1,2}) \wedge \\
& ((\mathbf{Y}T_{2,1}) \Rightarrow R_{2,1}) \wedge \\
& ((\mathbf{Y}T_{1,3}) \Rightarrow R_{1,3}) \wedge \\
& ((\mathbf{Y}T_{3,1}) \Rightarrow R_{3,1}) \wedge \\
& ((\mathbf{Y}T_{2,3}) \Rightarrow R_{2,3}) \wedge \\
& ((\mathbf{Y}T_{3,2}) \Rightarrow R_{3,2}))
\end{aligned}$$

$$\begin{aligned}
\phi_{TS5} \equiv \mathbf{XG} \quad & (((\mathbf{Y}(\neg M_0 \wedge \neg T_{0,1})) \Rightarrow (\neg T_{0,1} \wedge \neg R_{0,1})) \wedge \\
& ((\mathbf{Y}(\neg M_1 \wedge \neg T_{1,0})) \Rightarrow (\neg T_{1,0} \wedge \neg R_{1,0})) \wedge \\
& ((\mathbf{Y}(\neg M_0 \wedge \neg T_{0,2})) \Rightarrow (\neg T_{0,2} \wedge \neg R_{0,2})) \wedge \\
& ((\mathbf{Y}(\neg M_2 \wedge \neg T_{2,0})) \Rightarrow (\neg T_{2,0} \wedge \neg R_{2,0})) \wedge \\
& ((\mathbf{Y}(\neg M_1 \wedge \neg T_{1,2})) \Rightarrow (\neg T_{1,2} \wedge \neg R_{1,2})) \wedge \\
& ((\mathbf{Y}(\neg M_2 \wedge \neg T_{2,1})) \Rightarrow (\neg T_{2,1} \wedge \neg R_{2,1})) \wedge \\
& ((\mathbf{Y}(\neg M_1 \wedge \neg T_{1,3})) \Rightarrow (\neg T_{1,3} \wedge \neg R_{1,3})) \wedge \\
& ((\mathbf{Y}(\neg M_3 \wedge \neg T_{3,1})) \Rightarrow (\neg T_{3,1} \wedge \neg R_{3,1})) \wedge \\
& ((\mathbf{Y}(\neg M_2 \wedge \neg T_{2,3})) \Rightarrow (\neg T_{2,3} \wedge \neg R_{2,3})) \wedge \\
& ((\mathbf{Y}(\neg M_3 \wedge \neg T_{3,2})) \Rightarrow (\neg T_{3,2} \wedge \neg R_{3,2})))
\end{aligned}$$

$$\begin{aligned}
\phi_{round} \equiv \mathbf{G} \quad & \neg((T_{0,1} \vee T_{1,0} \vee T_{0,2} \vee T_{2,0} \vee T_{1,2} \vee T_{2,1} \vee T_{1,3} \vee T_{3,1} \vee T_{2,3} \vee T_{3,2}) \wedge \\
& (\neg R_{0,1} \wedge \neg R_{1,0} \wedge \neg R_{0,2} \wedge \neg R_{2,0} \wedge \neg R_{1,2} \wedge \neg R_{2,1} \wedge \neg R_{1,3} \\
& \wedge \neg R_{3,1} \wedge \neg R_{2,3} \wedge \neg R_{3,2}))
\end{aligned}$$

$$\begin{aligned}
\phi_{init} \equiv \quad & (M_0 \wedge \neg M_1 \wedge \neg M_2 \wedge \neg M_3) \wedge \\
& (\neg T_{0,1} \wedge \neg R_{0,1} \wedge \neg T_{1,0} \wedge \neg R_{1,0} \wedge \\
& \neg T_{0,2} \wedge \neg R_{0,2} \wedge \neg T_{2,0} \wedge \neg R_{2,0} \wedge \\
& \neg T_{1,2} \wedge \neg R_{1,2} \wedge \neg T_{2,1} \wedge \neg R_{2,1} \wedge \\
& \neg T_{1,3} \wedge \neg R_{1,3} \wedge \neg T_{3,1} \wedge \neg R_{3,1} \wedge \\
& \neg T_{2,3} \wedge \neg R_{2,3} \wedge \neg T_{3,2} \wedge \neg R_{3,2})
\end{aligned}$$

For the synchronous case, instantiating the  $\phi_m$  and  $\phi_i$  of the previous chapter yields:

$$\begin{aligned}
\phi_i \equiv \quad & (M_0^s \wedge \neg M_1^s \wedge \neg M_2^s \wedge \neg M_3^s) \wedge \\
& (\neg S_{0,1} \wedge \neg S_{1,0} \wedge \neg S_{0,2} \wedge \neg S_{2,0} \wedge \neg S_{1,2} \wedge \neg S_{2,1} \wedge \neg S_{1,3} \wedge \neg S_{3,1} \wedge \neg S_{2,3} \wedge \neg S_{3,2})
\end{aligned}$$



$$\begin{aligned}
\phi_m \equiv & \text{(\mathbf{XG}( (M_0^s \Leftrightarrow S_{1,0} \vee S_{2,0}) \wedge \\
& (M_1^s \Leftrightarrow S_{0,1} \vee S_{2,1} \vee S_{3,1}) \wedge \\
& (M_2^s \Leftrightarrow S_{0,2} \vee S_{1,2} \vee S_{3,2}) \wedge \\
& (M_3^s \Leftrightarrow S_{1,3} \vee S_{2,3})) \ ) \ \wedge} \\
& \text{(\mathbf{XG}( (S_{0,1} \Leftrightarrow \mathbf{Y}(M_0^s \wedge \neg S_{1,0})) \wedge} \\
& (S_{1,0} \Leftrightarrow \mathbf{Y}(M_1^s \wedge \neg S_{0,1})) \wedge \\
& (S_{2,0} \Leftrightarrow \mathbf{Y}(M_2^s \wedge \neg S_{0,2})) \wedge \\
& (S_{0,2} \Leftrightarrow \mathbf{Y}(M_0^s \wedge \neg S_{2,0})) \wedge \\
& (S_{1,2} \Leftrightarrow \mathbf{Y}(M_1^s \wedge \neg S_{2,1})) \wedge \\
& (S_{2,1} \Leftrightarrow \mathbf{Y}(M_2^s \wedge \neg S_{1,2})) \wedge \\
& (S_{1,3} \Leftrightarrow \mathbf{Y}(M_1^s \wedge \neg S_{3,1})) \wedge \\
& (S_{3,1} \Leftrightarrow \mathbf{Y}(M_3^s \wedge \neg S_{1,3})) \wedge \\
& (S_{2,3} \Leftrightarrow \mathbf{Y}(M_2^s \wedge \neg S_{3,2})) \wedge \\
& (S_{3,2} \Leftrightarrow \mathbf{Y}(M_3^s \wedge \neg S_{2,3}))) \ )}
\end{aligned}$$

By equation (6.8),(6.9) above, in order to demonstrate that no asynchronous run terminates before the synchronous run, we need to prove the following formula with the substitutions for  $\phi_{S2}$ ,  $\phi_{TS1}$ - $\phi_{TS5}$ ,  $\phi_{round}$ ,  $\phi_{init}$ ,  $\phi_i$  and  $\phi_m$ , as given above:

$$\begin{aligned}
& \neg( \phi_{S2} \wedge \phi_{TS1} \wedge \phi_{TS2} \wedge \phi_{TS3} \wedge \phi_{TS4} \wedge \phi_{TS5} \wedge \phi_{round} \wedge \phi_{init} \wedge \phi_m \wedge \phi_i \\
& \Rightarrow
\end{aligned}$$

$$\begin{aligned}
& \mathbf{F}( (\neg T_{0,1} \wedge \neg T_{1,0} \wedge \neg T_{0,2} \wedge \neg T_{2,0} \wedge \neg T_{1,2} \wedge \neg T_{2,1} \wedge \neg T_{1,3} \wedge \neg T_{3,1} \wedge \neg T_{2,3} \wedge \neg T_{3,2}) \wedge \\
& (\neg M_0 \wedge \neg M_1 \wedge \neg M_2 \wedge \neg M_3) \wedge (M_0^s \vee M_1^s \vee M_2^s \vee M_3^s) ) \ )
\end{aligned}$$

This proof has been carried out using NuSMV as shown in code in Appendix C Section C.2 and returns *True*, proving that no asynchronous run terminates before the synchronous run.

## 6.5 Conclusions

We have chosen a specification method for asynchronous network flooding that is directed at performing an analysis of the termination times for such systems. Taking our cue from the synchronous case, we decided to use the number of rounds as the measure of time to termination. This has meant giving a plausible definition of ‘rounds’ for the asynchronous case which corresponds to the accepted definition of rounds when asynchronous flooding executes as a synchronous system. As all messages that arrive ‘at the same time’ are aggregated into the same round, this has ruled out standard process calculi approaches such as CSP [1] and CCS

[2], which only allow two processes at a time to synchronize the sending and receiving of messages. Flooding would require a node (as a process) to be able to synchronize the sending and receiving of messages with many nodes (processes) in the same round. Thus, the notion of a round would be lost and it would be difficult to recover a count of rounds, each of which would comprise multiple sends and receives. Indeed, the model checker, NuSMV, has an asynchronous process description language which can be used if desired, but the interleaved model of concurrency that it implements would have the same problem with specifying rounds. We have used NuSMV purely as a temporal logic prover for direct specifications of temporal constraints. In contrast to CSP and CCS, the calculus of broadcasting systems (CBS) [82] does allow one-to-many communication, mainly for sending to all processes (nodes), but processes can only send one at a time. Petri nets [83] also allows many messages to be sent in one instant, but messages ('tokens') are either sent from all incoming edges ('arcs') or none.

In our previous work on synchronous flooding, considerable emphasis was placed on specifying constraints on graph topologies, thereby defining multiple topologies for a given set of network nodes so that termination properties could be proved for *all* topologies. Here, we have compared the termination times between an asynchronous and a synchronous network by testing the validity of a formula of the form (see equation (6.8),(6.9)):

$$\begin{aligned} & \neg(\Phi^a(\dots, T_{g,h}, \dots, R_{g,h}, \dots, M_g, \dots) \wedge \Phi^s(\dots, S_{g,h}, \dots, M_g, \dots)) \\ & \Rightarrow \Theta_{a \text{ then } s} \end{aligned} \tag{6.10}$$

where  $\Phi^a$  defines all (non-deterministic) asynchronous runs,  $\Phi^s$  defines a *unique* (deterministic) synchronous run, and  $\Theta_{a \text{ then } s}$  asserts that an asynchronous run terminates before the synchronous run. So, with the negation, the formula (6.10) states that no asynchronous runs can terminate before the synchronous. If the formula is *False*, i.e., not valid, there is some asynchronous run that can terminate before the (unique) synchronous run. If we added edge propositional variables  $E_{g,h}$  to  $\Phi_s$  so that multiple topologies and therefore many synchronous runs were defined by  $\Phi_s$ , then if the resulting formula:

$$\begin{aligned} & \neg\Phi^a(\dots, T_{g,h}, \dots, R_{g,h}, \dots, M_g, \dots) \wedge \Phi^s(\dots, E_{g,h}, \dots, S_{g,h}, \dots, M_g, \dots) \\ & \Rightarrow \Theta_{a \text{ then } s} \end{aligned} \tag{6.11}$$

returned *False*, that would mean that *some* asynchronous run would terminate before *some* synchronous run. If (6.11) returned *True*, it would mean that *all* synchronous runs terminate before *all* asynchronous runs. To prove that *some*

asynchronous run terminates before *all* synchronous runs would require a more expressive temporal logic such as *QPTL* [84], which is not supported by NuSMV and for which verification is much more problematic.

Finally, we have only specified a model of bounded asynchronous flooding where a sent message can be delayed one round in a non-deterministic manner. This could be extended to delays of up to a larger fixed number of rounds. Although different configurations of messages at nodes could result from the possibility of different lengths of delays, it is not clear whether the properties that we have focussed on in this work, namely the fewest rounds to the termination of a run, would be affected.

# Chapter 7

## Conclusions

This chapter provides a general set of conclusions for the research completed in this thesis. The individual chapters in the thesis contain more detailed conclusions, to which the reader is also referred.

Temporal logic has been used in specification and verification of properties of systems which have interacting components and environment. These systems are referred to as reactive systems. Safety-critical systems is a type of the reactive systems where safety is critical and fault tolerance is avoided as errors can cause loss of people lives and/or huge financial loss. This research focused on distributed systems network algorithms. Distributed systems have components that interact with each other and the environment. The use of linear temporal logic in this research showed how powerful is temporal logic in specifying properties of network algorithms. Both safety and liveness properties were specified and verified in this research. This work presented novel approaches to specify and verify these two important properties

This first part of this work handled transactions on data items being accessed in a concurrent manner. The concurrency of such a scenario is representative of distributed network systems where shared memory is used. Concurrent access of shared resources makes handling the different transactions a very difficult task. The protocol we presented has a certain similarity when the data items are shared resources. The data items are stored on routers' memories which are accessed by unlimited number of transactions in a concurrent manner. We presented a protocol that can be used to detect cycles caused by conflicting transactions accessing the shared resources. The use temporal logic and the model checker NuSMV to model the network of routers accessed by unlimited number of transactions according to the gap theory presented the first contribution novelty.

On the other hand, the other type of distributed network systems uses message passing. We modelled the well-known flooding algorithm, which uses message passing between different network nodes to accomplish its required tasks. The

message is sent from one (initial) node to all other nodes in the network. We specified the flooding algorithm using linear temporal logic and, using the NuSMV model checker, subsequently verified these specifications. We successfully specified the termination property for network flooding, where now we can determine if network flooding on a given topology will terminate or otherwise. This presented the second contribution novelty as researchers didn't consider using temporal logic to specify properties of the memory-less flooding algorithm. We also specified asynchronous flooding and compared its termination with the previously described synchronous flooding. This presented the third contribution in this research.

We can see that temporal logic can be used in the specification and verification of network algorithms. In the case that a property was not met, the model checker gives a counterexample showing the states that cause the error. The states represent a trace which can be helpful in defining the error. This provides some considerable benefit over other techniques where it is not otherwise possible to model such problems. Although temporal logic is powerful and model checkers have improved over the past few decades, the state explosion problem limits problems where there are a large number of states to be specified and verified. Even with the specifications presented in this research, and the use of one of the most powerful model checkers available, NuSMV, proofs will only be possible in practice for fairly small sizes of network. A strong mathematical background can help in solving and proving some problems where data is accessed in different ways other than what is presented in this work. Some problems faced during this research will need more time and effort to tackle in addition to higher mathematical skills.

## 7.1 Future work

The contribution of Chapter 4 has different potential applications. Due to the limitation of the order in which the transactions gain access to the data items, this research opens the door to further research in this area. Future work will consider other situations where data is accessed in a different manner.

The flooding problem gives an excellent example of a very basic algorithm on a network node - on receipt of a message, the node sends on the message to all its neighbours except for those from which it received the message. In the same way that network topologies can be easily modified by altering the topological constraints, future work could also investigate whether the code/algorithm at the nodes could be changed by supplying new, possibly more sophisticated, message-processing constraints which can then be verified.

In Chapter 6, we have only specified a model of bounded asynchronous flooding where a sent message can be delayed one round in a non-deterministic manner.

Future work could be extended to consider delays of up to a larger, fixed number of rounds. Although different configurations of the messages at the nodes could result from the possibility of different lengths of delays, it is not clear whether the properties that we have focussed on in this work, namely the fewest number of rounds to the termination of a run, would be affected.

Finally, future work could also attempt to develop tools to support the temporal logic analysis of distributed software/algorithms. For example, a tool could input pseudo-code for the algorithm at all nodes and output an appropriate NuSMV script for the network where each node has the behaviour of the algorithm.

**List of Publications:**

1. An Efficient Administrative Networking Protocol Specification and Verification Using Temporal Logic , International Journal of Computer Applications in Technology. Submitted paper.
2. Specification of Synchronous Network Flooding in Temporal Logic, International Arab Journal of Information Technology. Submitted paper.

# References

- [1] C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.
- [2] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.
- [3] “Samsung confirms battery faults as cause of note 7 fires,” Jan 2017.
- [4] D. Kroenke and D. J. Auer, *Database processing*. Prentice Hall, 2010.
- [5] C. Baier and J.-P. Katoen, “Principles of model checking, vol. 950,” 2008.
- [6] N. G. Leveson and C. S. Turner, “An investigation of the therac-25 accidents,” *Computer*, vol. 26, no. 7, pp. 18–41, 1993.
- [7] M. Wolf, “Embedded software in crisis,” *Computer*, vol. 49, pp. 88–90, Jan 2016.
- [8] <http://heartbeat.skype.com/2007/08/>. Accessed: 2016-09-10.
- [9] D. B. R. Mendick, “Worldwide airport chaos after computer check-in systems crash,” Sep 2017.
- [10] I. Sommerville, *Software Engineering*. 2010.
- [11] E. J. Braude and M. E. Bernstein, *Software engineering: modern approaches*. Waveland Press, 2016.
- [12] R. Alshorman and W. Hussak, “A serializability condition for multi-step transactions accessing ordered data,” *International Journal of Computer Science*, vol. 4, 01 2009.
- [13] S. N. R. Elmasri, *Fundamentals of Database Systems*. Addison-Wesley, fourth edition ed., 2004.
- [14] C. Papadimitriou, *The theory of database concurrency control*. Computer Science Press, 1986.



- [15] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.
- [16] C. Coronel, S. Morris, and P. Rob, “Database systems: design, implementation, and management,” *Cengage Learning*, vol. 9, 2009.
- [17] D. E. Comer and R. E. Droms, *Computer Networks and Internets*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 4 ed., 2003.
- [18] H. Attiya and J. Welch, *Distributed computing: fundamentals, simulations, and advanced topics*, vol. 19. John Wiley & Sons, 2004.
- [19] S. Kutten, G. Pandurangan, D. Peleg, P. Robinson, and A. Trehan, “On the complexity of universal leader election,” *J. ACM*, vol. 62, no. 1, pp. 7:1–7:27, 2015.
- [20] D. Peleg, “Distributed computing,” *SIAM Monographs on discrete mathematics and applications*, vol. 5, 2000.
- [21] M. Raynal, *Distributed algorithms for message-passing systems*, vol. 500. Springer, 2013.
- [22] J. Wan, D. Yuan, and X. Xu, “A review of routing protocols in wireless sensor networks,” in *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM'08. 4th International Conference on*, pp. 1–4, IEEE, 2008.
- [23] A. P. Zohar Manna Zohar Manna Zohar Manna, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1 ed., 1992.
- [24] M. Abadi and Z. Manna, “Temporal logic programming,” *J. Symb. Comput*, vol. 8, no. 3, pp. 277–295, 1989.
- [25] A. Sistla, “Safety, liveness and fairness in temporal logic,” *Formal Aspects of Computing*, vol. 6, no. 5, pp. 495–511, 1994.
- [26] L. Lamport, “Proving the correctness of multiprocess programs,” *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977.
- [27] H. Klapuri, J. Takala, and J. Saarinen, “Safety, liveness and real-time in embedded system design,” *Journal of Network and Computer Applications*, vol. 22, no. 2, pp. 69–89, 1999.
- [28] L. Lamport, “A new approach to proving the correctness of multiprocess programs,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 84–97, 1979.

- [29] F. C. Gartner, H. Pagnia, and H. Vogt, “Approaching a formal definition of fairness in electronic commerce,” in *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*, pp. 354–359, IEEE, 1999.
- [30] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, “Model checking and the state explosion problem,” in *Tools for Practical Software Verification*, pp. 1–30, Springer, 2012.
- [31] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.
- [32] M. Fisher, “A model checker for linear time temporal logic,” *Formal Aspects of Computing*, vol. 4, no. 3, pp. 299–319, 1992.
- [33] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Progress on the state explosion problem in model checking,” in *Informatics*, pp. 176–194, Springer, 2001.
- [34] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, “Symbolic model checking: 1020 states and beyond,” *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.
- [35] J. R. Burch, E. M. Clarke, and D. E. Long, *Symbolic model checking with partitioned transition relations*. Carnegie-Mellon University. Department of Computer Science, 1991.
- [36] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “Nusmv: A new symbolic model verifier,” in *International conference on computer aided verification*, pp. 495–499, Springer, 1999.
- [37] G. Holzmann and S. M. Checker, “The: Primer and reference manual,” *ISBN: 0-321-22862-6*, 2004.
- [38] G. Le Lann, “A methodology for designing and dimensioning critical complex computing systems,” in *Engineering of Computer-Based Systems, 1996. Proceedings., IEEE Symposium and Workshop on*, pp. 332–339, IEEE, 1996.
- [39] B. Long, P. Strooper, and L. Wildman, “A method for verifying concurrent java components based on an analysis of concurrency failures,” *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 281–294, 2007.
- [40] S. C. Cheung and J. Kramer, “Checking subsystem safety properties in compositional reachability analysis,” in *Proceedings of the 18th international conference on Software engineering*, pp. 144–154, IEEE Computer Society, 1996.

- [41] C. H. Papadimitriou, “The serializability of concurrent database updates,” *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.
- [42] F. Paternò and C. Santoro, “Integrating model checking and hci tools to help designers verify user interface properties,” in *International Workshop on Design, Specification, and Verification of Interactive Systems*, pp. 135–150, Springer, 2000.
- [43] A. Cauchi, G. Pace, and S. Spina, “Model checking user interfaces,” 2008.
- [44] M. B. Dwyer, V. Carr, and L. Hines, “Model checking graphical user interfaces using abstractions,” in *ACM SIGSOFT Software Engineering Notes*, vol. 22, pp. 244–261, Springer-Verlag New York, Inc., 1997.
- [45] P. Bellini, R. Mattolini, and P. Nesi, “Temporal logics for real-time system specification,” *ACM Computing Surveys (CSUR)*, vol. 32, no. 1, pp. 12–42, 2000.
- [46] S. Coogan and M. Arcaç, “Freeway traffic control from linear temporal logic specifications,” in *2014 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pp. 36–47, IEEE, 2014.
- [47] K.-Y. Lam, E. Chan, H.-W. Leung, and M.-W. Au, “Concurrency control strategies for ordered data broadcast in mobile computing systems,” *Information Systems*, vol. 29, no. 3, pp. 207–234, 2004.
- [48] S. Lim and H. Cho, “Timestamp based concurrency control in broadcast disks environment.,” in *AIS*, pp. 333–341, Springer, 2004.
- [49] K.-w. Lam, C. Wong, and W. Leung, “Using look-ahead protocol for mobile data broadcast,” in *Information Technology and Applications, 2005. ICITA 2005. Third International Conference on*, vol. 2, pp. 342–345, IEEE, 2005.
- [50] M.-P. Flé and G. Roucairol, “On serializability of iterated transactions,” in *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pp. 194–200, ACM, 1982.
- [51] D. Peled and A. Pnueli, “Proving partial order properties,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 143–182, 1994.
- [52] W. Hussak, “Serializable histories in quantified propositional temporal logic,” *International Journal of Computer Mathematics*, vol. 81, no. 10, pp. 1203–1211, 2004.

- [53] W. Hussak, “Specifying strict serializability of iterated transactions in propositional temporal logic,” *International Journal of Computer Science*, vol. 2, no. 2, pp. 150–156, 2007.
- [54] W. Hussak, “The serializability problem for a temporal logic of transaction queries,” *Journal of Applied Non-Classical Logics*, vol. 18, no. 1, pp. 67–78, 2008.
- [55] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [56] S. Hoory, N. Linial, and A. Wigderson, “Expander graphs and their applications,” *Bulletin of the AMS*, vol. 43, no. 04, pp. 439–562, 2006.
- [57] A. Fiat and J. Saia, “Censorship resistant peer-to-peer networks,” *Theory of Computing*, vol. 3, no. 1, pp. 1–23, 2007.
- [58] A. Fiat and J. Saia, “Censorship resistant peer-to-peer content addressable networks,” in *SODA*, pp. 94–103, 2002.
- [59] N. Pippenger and G. Lin, “Fault-tolerant circuit-switching networks,” *SIAM J. Discrete Math.*, vol. 7, no. 1, pp. 108–118, 1994.
- [60] C. Dwork, D. Peleg, N. Pippenger, and E. Upfal, “Fault tolerance in networks of bounded degree,” *SIAM J. Comput.*, vol. 17, no. 5, pp. 975–988, 1988.
- [61] E. Upfal, “Tolerating a linear number of faults in networks of bounded degree,” *Inf. Comput.*, vol. 115, no. 2, pp. 312–320, 1994.
- [62] V. King, J. Saia, V. Sanwalani, and E. Vee, “Towards secure and scalable computation in peer-to-peer networks,” in *FOCS*, pp. 87–98, 2006.
- [63] S. Kutten, G. Pandurangan, D. Peleg, P. Robinson, and A. Trehan, “Sublinear bounds for randomized leader election,” *Theor. Comput. Sci.*, vol. 561, pp. 134–143, 2015.
- [64] K. C. Hillel and H. Shachnai, “Partial information spreading with application to distributed maximum coverage,” in *PODC ’10: Proceedings of the 28th ACM symposium on Principles of distributed computing*, (New York, NY, USA), ACM, 2010.
- [65] G. Pandurangan, P. Raghavan, and E. Upfal, “Building low-diameter P2P networks,” in *FOCS*, pp. 492–499, 2001.

- [66] C. Law and K. Y. Siu, “Distributed construction of random expander networks,” in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, vol. 3, pp. 2133–2143 vol.3, 2003.
- [67] C. Gkantsidis, M. Mihail, and A. Saberi, “Random walks in peer-to-peer networks: Algorithms and evaluation,” *Performance Evaluation*, vol. 63(3), pp. 241–263, 2006.
- [68] G. Pandurangan and A. Trehan, “Xheal: a localized self-healing algorithm using expanders,” *Distributed Computing*, vol. 27, no. 1, pp. 39–54, 2014.
- [69] G. Pandurangan, P. Robinson, and A. Trehan, “DEX: self-healing expanders,” *Distributed Computing*, vol. 29, no. 3, pp. 163–185, 2016.
- [70] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.
- [71] L. Cardelli and A. D. Gordon, “Mobile ambients,” in *International Conference on Foundations of Software Science and Computation Structure*, pp. 140–155, Springer, 1998.
- [72] N. A. Lynch and M. R. Tuttle, “Hierarchical correctness proofs for distributed algorithms,” in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pp. 137–151, ACM, 1987.
- [73] R. Alshorman and W. Hussak, “Multi-step transactions specification and verification in a mobile database community,” in *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pp. 1–6, IEEE, 2008.
- [74] K. Sen, G. Rosu, and G. Agha, “Generating optimal linear temporal logic monitors by coinduction,” in *ASIAN*, vol. 2896, pp. 260–275, Springer, 2003.
- [75] A. Pnueli and Z. Manna, “Temporal verification of reactive systems-safety,” 1995.
- [76] R. Bani Abdelrahman and R. Alshorman, “An efficient administrative networking protocol specification and verification using temporal logic,” *International Journal of Computer Applications in Technology*.
- [77] R. Alshorman and H. Fawareh, “Reducing conflict graph of multi-step transactions accessing ordered data with gaps,” *Mathematics in Computer Science*, vol. 40, pp. 1–8, 06 2013.

- [78] A. Pnueli, *Current Trends in Concurrency. Overviews and Tutorials*, ch. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends, pp. 510–584. Springer-Verlag New York, Inc., 1986.
- [79] L. Lamport, “What good is temporal logic?,” in *IFIP congress*, vol. 83, pp. 657–668, 1983.
- [80] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “Nusmv: a new symbolic model checker,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.
- [81] R. Bani Abdelrahman, R. Alshorman, W. Hussak, and A. Trehan, “Specification of synchronous network flooding in temporal logic,” *International Arab Journal of Information Technology*.
- [82] K. Prasad, “A calculus of broadcasting systems,” *Science of Computer Programming*, vol. 25, pp. 285–327, 1995.
- [83] J.L.Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.
- [84] Y. Kesten and A. Pnueli, “A complete proof system for QPTL,” *Journal of Logic and Computation*, vol. 12, pp. 701–745, 2002.

# Appendix A

## Routing protocol encoding into LTL

□

We used MODULE `move(Tr,n,Ta,Tb,Tc)`. The variables `(Tr,n,Ta,Tb,Tc)` represent:

`Tr`: a transaction that is currently in process.

`n`: an integer indicating the number of the transaction.

`Ta, Tb, Tc`: other transactions that are waiting in the queue.

`T1,T2,T3, T4`: transactions number one, two, three and four.

`r1x1`: T1 reads item x1.

`w1x1`: T1 writes on item x1.

The code is shown here:

```
{MODULE move(Tr,n,Ta,Tb,Tc)
ASSIGN
next(Tr) := case
Tr= begin1 &n= 1 &(!(Tr=r1x3) & (!(Ta=r2x3)) & (!(Tb=r3x3)) ) : r1x3;
Tr= r1x3 &n= 1 : w1x3;
    Tr= w1x3 &n= 1 : r1x4;
    Tr= r1x4 &n= 1 : w1x4;
    Tr= w1x4 &n= 1 : end1;
    Tr= end1 : begin1;

    Tr= begin2 &n= 2 &(!(Tr=r2x3) & !(Ta=r1x3)) & (!(Tb=r3x3)) ) : r2x3;
    Tr= r2x3 &n= 2 : w2x3;
    Tr= w2x3 &n= 2 : r2x4;
    Tr= r2x4 &n= 2 : w2x4;
```

```

Tr= w2x4 &n= 2 : end2;
Tr= end2 : begin2;

Tr= begin3 &n= 3 &(!(Tr=r3x3) & (!(Ta=r1x3)) & (!(Tb=r2x3)) ) : r3x3;
Tr= r3x3 &n= 3 : w3x3;
Tr= w3x3 &n= 3 : r3x5;
Tr= r3x5 &n= 3 : w3x5;
Tr= w3x5 &n= 3 : end3;
Tr= end3 : begin3;

Tr= begin4 &n= 4 &(!(Tr=r4x4) & (!(Ta=r1x4)) & (!(Tb=r2x4)) ) : r4x4;
Tr= r4x4 &n= 4 : w4x4;
Tr= w4x4 &n= 4 : r4x5;
Tr= r4x5 &n= 4 : w4x5;
Tr= w4x5 &n= 4 : end4;
Tr= end4 : begin4;
TRUE : Tr;
esac;

MODULE main
  VAR
    T1 : {begin1,r1x3,w1x3,r1x4,w1x4,end1};
    T2 : {begin2,r2x3,w2x3,r2x4,w2x4,end2};
    T3 : {begin3,r3x3,w3x3,r3x5,w3x5,end3};
    T4 : {begin4,r4x4,w4x4,r4x5,w4x5,end4};
    x: process move(T1,1,T2,T3,T4);
    y: process move(T2,2,T1,T3,T4);
    z: process move(T3,3,T1,T2,T4);
    w: process move(T4,4,T1,T2,T3);

  ASSIGN
    init(T1) := begin1;
    init(T2) := begin2;
    init(T3) := begin3;
    init(T4) := begin4;
    FAIRNESS (T1=end1)
    FAIRNESS (T2=end2)
    FAIRNESS (T3=end3)
    FAIRNESS (T4=end4)

```



LTLSPEC F ( (T1=r1x3)->X(T1=w1x3)->X(T3=r3x3)->X(T3=w3x3)->(T2=r2x3)->  
 X(T2=w2x3)->X(T2=r2x4)->X(T2=w2x4)->X(T1=r1x4)->X(T1=w1x4)->X(T4=r4x4)->  
 X(T4=w4x4)->X(T4=r4x5)->X(T4=w4x5)->(T3=r3x5)->X(T3=w3x5))

--T1

LTLSPEC G (((T1=r1x4) & 0(T1=r1x3)) -> 0(T1=w1x3))

--T2

LTLSPEC G (((T2=r2x4) & 0(T2=r2x3)) -> 0(T2=w2x3))

--T3

LTLSPEC G (((T3=r3x5) & 0(T3=r3x5)) -> 0(T3=w3x3))

--T4

LTLSPEC G (((T4=r4x5) & 0(T4=r4x4)) -> 0(T4=w4x4))

--T1

LTLSPEC G (T1=r1x3 -> (F (T1=w1x3 & F (T1=r1x4))))

--T2

LTLSPEC G (T2=r2x3 -> (F (T2=w2x3 & F (T2=r2x4))))

--T3

LTLSPEC G (T3=r3x3 -> (F (T3=w3x3 & F (T3=r3x5))))

--T4

LTLSPEC G (T4=r4x4 -> (F (T4=w4x4 & F (T4=r4x5))))

---T1

LTLSPEC G ((T1=r1x3) -> 0(T1=begin1))

LTLSPEC G ((T1=w1x3) -> 0(T1=r1x3))

LTLSPEC G ((T1=r1x4) -> 0(T1=w1x3))

LTLSPEC G ((T1=w1x4) -> 0(T1=r1x4))

LTLSPEC G ((T1=end1) -> 0(T1=w1x4))

--T2

LTLSPEC G ((T2=r2x3) -> 0(T2=begin2))

LTLSPEC G ((T2=w2x3) -> 0(T2=r2x3))

LTLSPEC G ((T2=r2x4) -> 0(T2=w2x3))

LTLSPEC G ((T2=w2x4) -> 0(T2=r2x4))

LTLSPEC G ((T2=end2) -> 0(T2=w2x4))

--T3

LTLSPEC G ((T3=r3x3) -> 0(T3=begin3))

LTLSPEC G ((T3=w3x3) -> 0(T3=r3x3))

```

LTLSPEC G ((T3=r3x5) -> 0(T3=w3x3))
LTLSPEC G ((T3=w3x5) -> 0(T3=r3x5))
LTLSPEC G ((T3=end3) -> 0(T3=w3x5))
---T4
LTLSPEC G ((T4=r4x4) -> 0(T4=begin4))
LTLSPEC G ((T4=w4x4) -> 0(T4=r4x4))
LTLSPEC G ((T4=r4x5) -> 0(T4=w4x4))
LTLSPEC G ((T4=w4x5) -> 0(T4=r4x5))
LTLSPEC G ((T4=end4) -> 0(T4=w4x5))

--T1
LTLSPEC G ((T1=begin1) -> X!((T1=r1x3)&(T1=w1x3)))
LTLSPEC G ((T1=r1x3) -> X!((T1=w1x3)&(T1=r1x4)))
LTLSPEC G ((T1=r1x4) -> X!((T1=w1x4)&(T1=end1)))
--T2
LTLSPEC G ((T2=begin2) -> X!((T2=r2x3)&(T2=w2x3)))
LTLSPEC G ((T2=r2x3) -> X!((T2=w2x3)&(T1=r2x4)))
LTLSPEC G ((T2=r2x4) -> X!((T2=w2x4)&(T2=end2)))
--T3
LTLSPEC G ((T3=begin3) -> X!((T3=r3x3)&(T3=w3x3)))
LTLSPEC G ((T3=r3x3) -> X!((T3=w3x3)&(T3=r3x5)))
LTLSPEC G ((T3=r3x5) -> X!((T3=w3x5)&(T3=end3)))
--T4
LTLSPEC G ((T4=begin4) -> X!((T4=r4x4)&(T4=w4x4)))
LTLSPEC G ((T4=r4x4) -> X!((T4=w4x4)&(T4=r4x5)))
LTLSPEC G ((T4=r4x5) -> X!((T4=w4x5)&(T4=end4)))
once by a transaction
LTLSPEC G ((T1=w1x3 & 0(T1=r1x3)) -> (F!(T1=r1x3)))
LTLSPEC G ((T1=w1x4 & 0(T1=r1x4)) -> (F!(T1=r1x4)))
LTLSPEC G ((T2=w2x3 & 0(T2=r2x3)) -> (F!(T2=r2x3)))
LTLSPEC G ((T2=w2x4 & 0(T2=r2x4)) -> (F!(T2=r2x4)))
LTLSPEC G ((T3=w3x3 & 0(T3=r3x3)) -> (F!(T3=r3x3)))
LTLSPEC G ((T3=w3x5 & 0(T3=r3x5)) -> (F!(T3=r3x5)))
LTLSPEC G ((T4=w4x4 & 0(T4=r4x4)) -> (F!(T4=r4x4)))
LTLSPEC G ((T4=w4x5 & 0(T4=r4x5)) -> (F!(T4=r4x5)))

--T1
LTLSPEC G ((T1=r1x3) -> F(T1=r1x3))

```

```

LTLSPEC G ((T1=w1x3) -> F(T1=w1x3))
LTLSPEC G ((T1=r1x4) -> F(T1=r1x4))
LTLSPEC G ((T1=w1x4) -> F(T1=w1x4))
--T2
LTLSPEC G ((T2=r2x3) -> F(T2=r2x3))
LTLSPEC G ((T2=w2x3) -> F(T2=w2x3))
LTLSPEC G ((T2=r2x4) -> F(T2=r2x4))
LTLSPEC G ((T2=w2x4) -> F(T2=w2x4))
--T3
LTLSPEC G ((T3=r3x3) -> F(T3=r3x3))
LTLSPEC G ((T3=w3x3) -> F(T3=w3x3))
LTLSPEC G ((T3=r3x5) -> F(T3=r3x5))
LTLSPEC G ((T3=w3x5) -> F(T3=w3x5))
--T4
LTLSPEC G ((T4=r4x4) -> F(T4=r4x4))
LTLSPEC G ((T4=w4x4) -> F(T4=w4x4))
LTLSPEC G ((T4=r4x5) -> F(T4=r4x5))
LTLSPEC G ((T4=w4x5) -> F(T4=w4x5))

LTLSPEC G ((T1=w1x3) -> O!(T2=r2x3))
LTLSPEC G ((T1=w1x3) -> O!(T3=r3x3))
LTLSPEC G ((T3=w3x3) -> O!(T2=r2x3))

LTLSPEC G ((T2=w2x4) -> O!(T1=r1x4))
LTLSPEC G ((T2=w2x4) -> O!(T3=r4x4))
LTLSPEC G ((T1=w1x4) -> O!(T4=r4x4))

LTLSPEC G ((T4=w4x5) -> O!(T3=r3x5))
-- We can also check in another way as follows:
--X3
LTLSPEC G ((T2=r2x3 & O(T1=r1x3)) -> (F!(T1=w1x3)))
LTLSPEC G ((T3=r3x3 & O(T1=r1x3)) -> (F!(T1=w1x3)))
LTLSPEC G ((T3=r3x3 & O(T2=r2x3)) -> (F!(T2=w2x3)))
--X4
LTLSPEC G ((T2=r2x4 & O(T1=r1x4)) -> (F!(T1=w1x4)))
LTLSPEC G ((T4=r4x4 & O(T1=r1x4)) -> (F!(T1=w1x4)))
LTLSPEC G ((T4=r4x4 & O(T2=r2x4)) -> (F!(T2=w2x4)))
--X5

```

```
LTLSPEC G ((T4=r4x5 & 0(T3=r3x5)) -> (F!(T3=w3x5)))
```

-----The gap G=1 tis means that we can have a cycle of length= 3

--- Here we check for this cycle only.

---these create the cycle if we put ! before G

--1321

```
LTLSPEC !G((T1=r1x3)->F(T3=w3x3)->F(T2=w2x3)->F(T1=w1x4))
```

--if we remove "!" before G it will execute and give TRUE , i.e,

--doesn't check for cycle.

--- this is continuing to check of n length cycle

--1241

```
LTLSPEC G((T1=r1x3)->F(T2=w2x3)->F(T4=w4x4)->F(T1=w1x4))
```

--2141

```
LTLSPEC G((T2=r2x3)->F(T1=w1x3)->F(T4=w4x4)->F(T2=w2x4))
```

--3242

```
LTLSPEC G((T3=r3x3)->F(T2=w2x3)->F(T4=w4x4)->F(T3=w3x5))
```

--3143

```
LTLSPEC G((T3=r3x3)->F(T1=w1x3)->F(T4=w4x4)->F(T3=w3x5))
```

--4234

```
LTLSPEC G((T4=r4x4)->F(T2=w2x4)->F(T3=w3x5)->F(T4=w4x5))
```

FAIRNESS running

# Appendix B

## Synchronous Flooding Algorithm encoding into LTL

□

### B.1 Topology1 terminates before Topology2 with initial node =0

□

```
MODULE main
VAR
-- Topology #1 nodes:
n1_0: node(TRUE);
n1_1: node(FALSE);
n1_2: node(FALSE);
n1_3: node(FALSE);
n1_4: node(FALSE);
-- Topology #1 edges:
e1_01: edge(n1_0, n1_1);
e1_02: edge(n1_0, n1_2);
e1_03: edge(n1_0, n1_3);
e1_04: edge(n1_0, n1_4);
e1_14: edge(n1_1, n1_4);
e1_23: edge(n1_2, n1_3);
-- Topology #2 nodes:
n2_0: node(TRUE);
```

```

n2_1: node(FALSE);
n2_2: node(FALSE);
n2_3: node(FALSE);
n2_4: node(FALSE);
-- Topology #2 edges:
e2_01: edge(n2_0, n2_1);
e2_04: edge(n2_0, n2_4);
e2_12: edge(n2_1, n2_2);
e2_23: edge(n2_2, n2_3);
e2_34: edge(n2_3, n2_4);
ASSIGN
-- Topology #1 rules:
next(n1_0.has_message) := next(e1_01.send_b_to_a | e1_02.send_b_to_a |
e1_03.send_b_to_a | e1_04.send_b_to_a);
next(n1_1.has_message) := next(e1_01.send_a_to_b | e1_14.send_b_to_a);
next(n1_2.has_message) := next(e1_02.send_a_to_b | e1_23.send_b_to_a);
next(n1_3.has_message) := next(e1_03.send_a_to_b | e1_23.send_a_to_b);
next(n1_4.has_message) := next(e1_04.send_a_to_b | e1_14.send_a_to_b);
-- Topology #2 rules:
next(n2_0.has_message) := next(e2_01.send_b_to_a | e2_04.send_b_to_a);
next(n2_1.has_message) := next(e2_01.send_a_to_b | e2_12.send_b_to_a);
next(n2_2.has_message) := next(e2_12.send_a_to_b | e2_23.send_b_to_a);
next(n2_3.has_message) := next(e2_23.send_a_to_b | e2_34.send_b_to_a);
next(n2_4.has_message) := next(e2_04.send_a_to_b | e2_34.send_a_to_b);
LTLSPEC F (
(!n1_0.has_message & !n1_1.has_message & !n1_2.has_message &
!n1_3.has_message & !n1_4.has_message) &
(n2_0.has_message | n2_1.has_message |
n2_2.has_message | n2_3.has_message | n2_4.has_message)
);

MODULE node(has_message_initially)
VAR
has_message: boolean;
ASSIGN
init(has_message) := has_message_initially;

MODULE edge(node_a, node_b)

```

```

VAR
send_a_to_b: boolean;
send_b_to_a: boolean;
ASSIGN
init(send_a_to_b) := FALSE;
init(send_b_to_a) := FALSE;
next(send_a_to_b) := node_a.has_message & !send_b_to_a;
next(send_b_to_a) := node_b.has_message & !send_a_to_b;

```

## B.2 Topology2 terminates before Topology1 with initial node =0

□

```

MODULE main
VAR
-- Topology #1 nodes:
n1_0: node(TRUE);
n1_1: node(FALSE);
n1_2: node(FALSE);
n1_3: node(FALSE);
n1_4: node(FALSE);
-- Topology #1 edges:
e1_01: edge(n1_0, n1_1);
e1_02: edge(n1_0, n1_2);
e1_03: edge(n1_0, n1_3);
e1_04: edge(n1_0, n1_4);
e1_14: edge(n1_1, n1_4);
e1_23: edge(n1_2, n1_3);
-- Topology #2 nodes:
n2_0: node(TRUE);
n2_1: node(FALSE);
n2_2: node(FALSE);
n2_3: node(FALSE);
n2_4: node(FALSE);
-- Topology #2 edges:

```

```

e2_01: edge(n2_0, n2_1);
e2_04: edge(n2_0, n2_4);
e2_12: edge(n2_1, n2_2);
e2_23: edge(n2_2, n2_3);
e2_34: edge(n2_3, n2_4);
ASSIGN
-- Topology #1 rules:
next(n1_0.has_message) := next(e1_01.send_b_to_a | e1_02.send_b_to_a |
e1_03.send_b_to_a | e1_04.send_b_to_a);
next(n1_1.has_message) := next(e1_01.send_a_to_b | e1_14.send_b_to_a);
next(n1_2.has_message) := next(e1_02.send_a_to_b | e1_23.send_b_to_a);
next(n1_3.has_message) := next(e1_03.send_a_to_b | e1_23.send_a_to_b);
next(n1_4.has_message) := next(e1_04.send_a_to_b | e1_14.send_a_to_b);
-- Topology #2 rules:
next(n2_0.has_message) := next(e2_01.send_b_to_a | e2_04.send_b_to_a);
next(n2_1.has_message) := next(e2_01.send_a_to_b | e2_12.send_b_to_a);
next(n2_2.has_message) := next(e2_12.send_a_to_b | e2_23.send_b_to_a);
next(n2_3.has_message) := next(e2_23.send_a_to_b | e2_34.send_b_to_a);
next(n2_4.has_message) := next(e2_04.send_a_to_b | e2_34.send_a_to_b);

LTLSPEC F (
(!n2_0.has_message & !n2_1.has_message & !n2_2.has_message &
!n2_3.has_message & !n2_4.has_message) &
(n1_0.has_message | n1_1.has_message | n1_2.has_message |
n1_3.has_message | n1_4.has_message)
);

MODULE node(has_message_initially)
VAR
has_message: boolean;
ASSIGN
init(has_message) := has_message_initially;

MODULE edge(node_a, node_b)
VAR
send_a_to_b: boolean;
send_b_to_a: boolean;
ASSIGN
init(send_a_to_b) := FALSE;

```



```

init(send_b_to_a) := FALSE;
next(send_a_to_b) := node_a.has_message & !send_b_to_a;
next(send_b_to_a) := node_b.has_message & !send_a_to_b;

```

### B.3 Topology1 terminates before Topology2 regardless of initial node

[]

```

--- This code checks if topology1 terminates before
--- topology2 regardless of starting state

```

```

MODULE main

```

```

VAR

```

```

t1: topology1;

```

```

t2: topology2;

```

```

LTLSPEC F (t1.is_terminated & !t2.is_terminated);

```

```

--LTLSPEC F (t2.is_terminated & !t1.is_terminated);

```

```

MODULE topology1

```

```

VAR

```

```

n0: node;

```

```

n1: node;

```

```

n2: node;

```

```

n3: node;

```

```

n4: node;

```

```

e01: edge(n0, n1);

```

```

e02: edge(n0, n2);

```

```

e03: edge(n0, n3);

```

```

e04: edge(n0, n4);

```

```

e14: edge(n1, n4);

```

```

e23: edge(n2, n3);

```

```

INIT

```

```

count(n0.has_message, n1.has_message, n2.has_message, n3.has_message,
n4.has_message) = 1;

```

```

ASSIGN

```

```

next(n0.has_message) := next(e01.send_b_to_a | e02.send_b_to_a |

```

```

e03.send_b_to_a | e04.send_b_to_a);
next(n1.has_message) := next(e01.send_a_to_b | e14.send_b_to_a);
next(n2.has_message) := next(e02.send_a_to_b | e23.send_b_to_a);
next(n3.has_message) := next(e03.send_a_to_b | e23.send_a_to_b);
next(n4.has_message) := next(e04.send_a_to_b | e14.send_a_to_b);
DEFINE
is_terminated := !n0.has_message & !n1.has_message & !n2.has_message &
!n3.has_message & !n4.has_message;

MODULE topology2
VAR
n0: node;
n1: node;
n2: node;
n3: node;
n4: node;
e01: edge(n0, n1);
e04: edge(n0, n4);
e12: edge(n1, n2);
e23: edge(n2, n3);
e34: edge(n3, n4);
INIT
count(n0.has_message, n1.has_message, n2.has_message, n3.has_message,
n4.has_message) = 1;
ASSIGN
next(n0.has_message) := next(e01.send_b_to_a | e04.send_b_to_a);
next(n1.has_message) := next(e01.send_a_to_b | e12.send_b_to_a);
next(n2.has_message) := next(e12.send_a_to_b | e23.send_b_to_a);
next(n3.has_message) := next(e23.send_a_to_b | e34.send_b_to_a);
next(n4.has_message) := next(e04.send_a_to_b | e34.send_a_to_b);
DEFINE
is_terminated := !n0.has_message & !n1.has_message & !n2.has_message &
!n3.has_message & !n4.has_message;

MODULE node
VAR
has_message: boolean;

MODULE edge(node_a, node_b)

```

*APPENDIX B. SYNCHRONOUS FLOODING ALGORITHM ENCODING INTO LTL120*

VAR

send\_a\_to\_b: boolean;

send\_b\_to\_a: boolean;

ASSIGN

init(send\_a\_to\_b) := FALSE;

init(send\_b\_to\_a) := FALSE;

next(send\_a\_to\_b) := node\_a.has\_message & !send\_b\_to\_a;

next(send\_b\_to\_a) := node\_b.has\_message & !send\_a\_to\_b;

# Appendix C

## Asynchronous flooding algorithm encoding into LTL

□

### C.1 Asynchronous Can Terminate Before Synchronous

□

```
MODULE main
VAR
-- Declare models:
async: model_with_3_nodes(0);
sync: model_with_3_nodes(0);
ASSIGN
-- Define edges of async model:
init(async.e01.__mode) := asynchronous;
init(async.e02.__mode) := asynchronous;
init(async.e12.__mode) := asynchronous;
-- Define edges of sync model:
init(sync.e01.__mode) := synchronous;
init(sync.e02.__mode) := synchronous;
init(sync.e12.__mode) := synchronous;
LTLSPEC
!F (async.is_terminated & !sync.is_terminated);

MODULE model_with_3_nodes(initially_active_node)
```

```

VAR
n0: node(initially_active_node = 0);
n1: node(initially_active_node = 1);
n2: node(initially_active_node = 2);
e01: maybe_edge(n0, n1);
e02: maybe_edge(n0, n2);
e12: maybe_edge(n1, n2);
ASSIGN
next(n0.has_message) := next(e01.b_to_a.received | e02.b_to_a.received);
next(n1.has_message) := next(e01.a_to_b.received | e12.b_to_a.received);
next(n2.has_message) := next(e02.a_to_b.received | e12.a_to_b.received);
DEFINE
is_terminated := !n0.has_message & !n1.has_message & !n2.has_message;
INVAR
!is_terminated | !(
e01.__has_transmitting |
e02.__has_transmitting |
e12.__has_transmitting
);
JUSTICE
TRUE;

MODULE node(has_message_initially)
VAR
has_message: boolean;
ASSIGN
init(has_message) := has_message_initially;

MODULE maybe_edge(node_a, node_b)
FROZENVAR
__mode: {disabled, synchronous, asynchronous};
VAR
a_to_b: maybe_directed_subedge(node_a, __mode, b_to_a);
b_to_a: maybe_directed_subedge(node_b, __mode, a_to_b);
DEFINE
__has_transmitting := a_to_b.transmitting | b_to_a.transmitting;

MODULE maybe_directed_subedge(start, mode, reverse)
VAR

```

```

transmitting: boolean;
received: boolean;
DEFINE
__should_send := (mode != disabled) & start.has_message & !reverse.received;
ASSIGN
init(transmitting) := FALSE;
init(received) := FALSE;
next(transmitting) := case
!__should_send : FALSE;
transmitting : TRUE;
TRUE : {FALSE, mode = asynchronous};
esac;
next(received) := __should_send xor transmitting xor next(transmitting);

```

## C.2 Asynchronous Cannot Terminate Before Synchronous

□

--Asynchronous Cannot Terminate Before Synchronous

```

MODULE main
VAR
-- Declare models:
async: model_with_4_nodes(0);
sync: model_with_4_nodes(0);
ASSIGN
-- Define edges of async model:
init(async.e01.__mode) := asynchronous;
init(async.e02.__mode) := asynchronous;
init(async.e03.__mode) := disabled;
init(async.e12.__mode) := asynchronous;
init(async.e13.__mode) := asynchronous;
init(async.e23.__mode) := asynchronous;
-- Define edges of sync model:
init(sync.e01.__mode) := synchronous;
init(sync.e02.__mode) := synchronous;

```

```

init(sync.e03.__mode) := disabled;
init(sync.e12.__mode) := synchronous;
init(sync.e13.__mode) := synchronous;
init(sync.e23.__mode) := synchronous;
LTLSPEC
!F (async.is_terminated & !sync.is_terminated);

MODULE model_with_4_nodes(initially_active_node)
VAR
n0: node(initially_active_node = 0);
n1: node(initially_active_node = 1);
n2: node(initially_active_node = 2);
n3: node(initially_active_node = 3);
e01: maybe_edge(n0, n1);
e02: maybe_edge(n0, n2);
e03: maybe_edge(n0, n3);
e12: maybe_edge(n1, n2);
e13: maybe_edge(n1, n3);
e23: maybe_edge(n2, n3);
ASSIGN
next(n0.has_message) := next(e01.b_to_a.received | e02.b_to_a.received |
e03.b_to_a.received);
next(n1.has_message) := next(e01.a_to_b.received | e12.b_to_a.received |
e13.b_to_a.received);
next(n2.has_message) := next(e02.a_to_b.received | e12.a_to_b.received |
e23.b_to_a.received);
next(n3.has_message) := next(e03.a_to_b.received | e13.a_to_b.received |
e23.a_to_b.received);
DEFINE
is_terminated := !n0.has_message & !n1.has_message & !n2.has_message &
!n3.has_message;
INVAR
!is_terminated | !(
e01.__has_transmitting |
e02.__has_transmitting |
e03.__has_transmitting |
e12.__has_transmitting |
e13.__has_transmitting |
e23.__has_transmitting

```

```

);
JUSTICE
TRUE;

MODULE node(has_message_initially)
VAR
has_message: boolean;
ASSIGN
init(has_message) := has_message_initially;

MODULE maybe_edge(node_a, node_b)
FROZENVAR
__mode: {disabled, synchronous, asynchronous};
VAR
a_to_b: maybe_directed_subedge(node_a, __mode, b_to_a);
b_to_a: maybe_directed_subedge(node_b, __mode, a_to_b);
DEFINE
__has_transmitting := a_to_b.transmitting | b_to_a.transmitting;

MODULE maybe_directed_subedge(start, mode, reverse)
VAR
transmitting: boolean;
received: boolean;
DEFINE
__should_send := (mode != disabled) & start.has_message & !reverse.received;
ASSIGN
init(transmitting) := FALSE;
init(received) := FALSE;
next(transmitting) := case
!__should_send : FALSE;
transmitting : TRUE;
TRUE : {FALSE, mode = asynchronous};
esac;
next(received) := __should_send xor transmitting xor next(transmitting);

```