

SEMANTICS OF AN OPTIONAL PARALLEL OPERATOR FOR CSP

by

Theunis J. Steyn

Submitted in partial fulfilment of the requirements for the degree

Master of Engineering (Computer Engineering)

in the

Department of Electrical, Electronic and Computer Engineering
Faculty of Engineering, Built Environment and Information Technology

UNIVERSITY OF PRETORIA

June 2015

SUMMARY

SEMANTICS OF AN OPTIONAL PARALLEL OPERATOR FOR CSP

by

Theunis J. Steyn

Supervisor(s): Prof. S. Gruner and Prof. G.P. Hancke
Department: Electrical, Electronic and Computer Engineering
University: University of Pretoria
Degree: Master of Engineering (Computer Engineering)
Keywords: Optional Parallelism, Communicating Sequential Processes, Wireless Sensor Networks, Operational Semantics, Formal Modelling, Broadcasting, Directional Synchronisation, Optional Parallelism Translation, OptoCSP, OpTrace

Communicating Sequential Processes (CSP) is arguably one of the most widely used process algebras. It has been extensively studied and expanded since its inception in the late 1970s. One of the fundamental assumptions of parallelism in CSP is that all processes have to jointly engage in synchronised events. There are cases however, especially when Wireless Sensor Networks (WSN) are modelled in CSP, where this restriction constrains the expressive capacity of CSP from a practical perspective. Optional parallelism lifts the restriction of parallelism in CSP by allowing processes to partially engage in synchronisation events. WSNs often have node or communication failures which increases the complexity of the CSP specifications of such WSNs. Basic communication constructs like broadcasting are also difficult to model in CSP, and other process algebras have been developed to allow broadcasting communication. Optional parallelism reduces the complexity by allowing processes to broadcast to other processes as well as to opt out of synchronisation when they are not ready to synchronise. The notion of optional parallelism introduces a new operator to CSP without redefining the semantics of existing operators or the definition of a new process algebra entirely.

This dissertation details the design of a translation of the definition of optional parallelism into classical CSP operators which will enable optional parallelism to easily be used in existing

CSP model-checkers. The core of the translation entails the addition of a channel modelling artefact to the existing process definitions which allows a communication event between the process and its environment to occur or not. The processes are thus independent from each other with the channel artefacts orchestrating the synchronisation between them. If a process is not ready to engage in an event, its channel ignores the event, and the process essentially opts out of synchronisation. Different combinations of the channel artefacts result in different directional synchronisation cases. Simplex synchronisation has the same behaviour as broadcasting. Duplex communication constructs are also defined which has the same behaviour as bidirectional synchronisation. Focus was given on the operational semantics of optional parallelism in the development of a classical CSP translation. This resulted in a new optional parallel operator being defined with the same operational semantics as the initially defined optional parallel operator. The operational semantics of the new optional parallel operator have been expanded to allow for directional synchronisation which better defines the notion of broadcasting and bidirectional communication links.

Two software tools were developed, OptoCSP and OpTrace. OptoCSP provides the functionality to convert CSP system definitions containing the optional parallel operator into CSP definitions containing only classical CSP operators. The software tools developed greatly simplified the trace generation of systems containing the optional parallel operator as this had to be manually scrutinised by hand. OpTrace was used to test for trace refinement between the initial optional parallel definitions and the translation of this dissertation with the use of WSN graph structure definitions. It has a trace generator which has the CSP step laws of the initial optional parallel operator implemented and a model generator which converts the WSN graph structures into CSP models with the new optional parallel translation. The model-checking tool, ProCSP, is used together with OpTrace to perform the trace assertions of the computed traces and the models. OpTrace and ProCSP was used to conclude if the test scenarios presented in this dissertation passed or not.

A relationship between the traces of WSN systems defined with the initial optional parallel operator and the translation thereof has been found. This result enables the optional parallel translation to be used as a macro for systems of optional parallelism to be model-checked with existing model-checkers. Optional parallelism has been studied in depth in the traces domain of CSP and the work of this dissertation shows that optional parallelism can be applied in the field of WSNs, as well as to define systems with optional parallel behaviour by using classical

CSP operators. It is therefore a new construct which can be used with existing operational semantics of the operators of CSP.

OPSOMMING

SEMANTIEK VAN 'N OPSIONELE PARALLELLE OPERATEUR VIR KSP

deur

Theunis J. Steyn

Studieleier(s):	Prof. S. Gruner en Prof. G.P. Hancke
Departement:	Elektriese, Elektroniese en Rekenaar-Ingenieurswese
Universiteit:	Universiteit van Pretoria
Graad:	Magister in Ingenieurswese (Rekenaaringenieurswese)
Sleutelwoorde:	Opsionele Parallelisme, Kommunikerende Sekwensiële Prosesse, Draadlose Sensor Netwerke, Operasionele Semantiek, Formele Modelling, Uitsaaikommunikasie, Direksionele Sinchronisasie, Opsionele Parallellisme Vertaling, OptoCSP, OpTrace

Kommunikerende Sekwensiële Prosesse (KSP) is waarskynlik een van die mees algemene gebruikte prosesalgebras. Dit is omvattend gebestudeer en uitgebrei sedert sy ontstaan in die laat 1970s. Een van die fundamentele aannames van parallelisme in KSP is dat al die prosesse gesamentlik betrokke moet raak in gesinchroniseerde gebeure. Daar is egter gevalle, veral wanneer Draadlose Sensor Netwerke (DSN) in KSP gemodelleer word, waar dit die beeldende vermoë van KSP vanuit 'n praktiese oogpunt beperk. Opsionele parallelisme lig die beperking van parallelisme in KSP deur prosesse toe te laat om gedeeltelik betrokke te raak in sinchronisasie. DSNs het dikwels node- of kommunikasiemislukkings wat die kompleksiteit van die KSP spesifikasies van sodanige DSNs verhoog. Basiese kommunikasiebeginsels soos uitsaai is moeilik om te modelleer in KSP en ander prosesalgebras is ontwikkel om voorsiening te maak vir uitsaaikommunikasie. Opsionele parallelisme verminder die kompleksiteit van modelle deur toe te laat dat prosesse uitsaai aan ander prosesse. Prosesse word ook toegelaat om self te kies om uit 'n sinchronisasie uitgesluit te wees wanneer hulle nie gereed is om te sinchroniseer nie. Die idee van opsionele parallelisme stel 'n nuwe operateur voor vir KSP sonder die herdefiniëring van die semantiek van bestaande operateurs of die definisie van 'n

nuwe prosesalgebra in geheel.

Hierdie verhandeling beskryf die ontwerp van 'n vertaling van die definisie van opsionele parallelisme in klassieke KSP operateurs wat opsionele parallelisme in staat stel om maklik gebruik te word in bestaande KSP modelkontroleerders. Die kern van die vertaling behels die toevoeging van 'n atomiese kanaalartefak aan die bestaande prosesdefinisies wat toelaat dat 'n kommunikasie gebeurtenis tussen die proses en die omgewing voorkom of nie. Die prosesse is dus onafhanklik van mekaar met die kanaal artefakte wat die kommunikasie tussen hulle beheer. As 'n proses nie gereed is om betrokke te raak in 'n gebeurtenis nie, ignoreer sy kanaal artefak die gebeurtenis, en die proses kies om uit die sinchronisasie gebeurtenis uitgesluit te wees. Verskillende kombinasies van die kanaalartefakte lei tot verskillende sinchronisasie gevalle wat rigting aandui. Simplekse sinchronisasie het dieselfde gedrag as uitsaai-kommunikasie. Duplekse kommunikasie beginsels word ook gedefinieer wat dieselfde gedrag as tweerigting sinchronisasie het. Fokus is gegee aan die operasionele semantiek van opsionele parallelisme in die ontwikkeling van 'n klassieke KSP vertaling. Dit het gelei tot 'n nuwe opsionele parallelle operateur wat gedefinieer is met dieselfde operasionele semantiek as die aanvanklike definisie van die opsionele parallelle operateur. Die operasionele semantiek van die nuwe opsionele parallelle operateur is uitgebrei om voorsiening te maak vir die rigting sinchronisasie wat die idee van uitsaai en tweerigting kommunikasieskakels beter definieer.

Twee sagteware programme is ontwikkel, OptoCSP en OpTrace. OptoCSP bied die funksie om KSP stelseldefinisies met die opsionele parallelle operateur te omskep in KSP definisies wat slegs klassieke KSP operateurs bevat. Die ontwikkelde programmatuur vereenvoudig die berekening van die gebeurtenisse van stelsels wat die opsionele parallelle operateur bevat, aangesien dit tans onder die loop geneem moet word met die hand. OpTrace is gebruik om te toets vir gebeurtenisverfyning tussen die aanvanklike opsionele parallelle definisie en die vertaling in hierdie verhandeling met die gebruik van DSN grafiekstruktuur definisies. Dit het 'n module wat gebeurtenisse bereken met behulp van die KSP stapwette van die aanvanklike definisie van die opsionele parallelle operateur. OpTrace het ook 'n modelbouer module wat die DSN grafiek strukture in KSP modelle, met die nuwe opsionele parallelle vertaling, omskep. Die modelkontrole programmatuur, ProCSP, word saam met OpTrace gebruik om die gebeurtenisse wat die KSP stelsels opwek te kontroleer teen hul definisies. OpTrace is gebruik om tot die gevolgtrekking te kom of die toets scenario's, wat in hierdie verhandeling beskryf is, geslaag het of nie.

'n Verband tussen die gebeurtenisse van DSN stelsels gedefinieer met die aanvanklike opsionele parallelle operateur en die vertaling daarvan is gevind. Hierdie resultaat stel die opsionele parallelle vertaling in staat om gebruik te word as 'n makro vir stelsels wat opsionele parallelisme bevat en om hul modelle te kontroleer met bestaande modelkontroleerders. Opsionele parallelisme is in diepte bestudeer in die gebeurtenis domein van KSP. Die werk van hierdie verhandeling toon dat opsionele parallelisme toegepas kan word in die gebied van DSNs sowel as in stelsels met opsionele parallelle gedragte deur die gebruik van klassieke KSP operateurs. Dit is dus 'n nuwe ontwikkeling wat gebruik kan word met die bestaande operasionele semantiek van die operateurs van KSP.

ACKNOWLEDGEMENTS

First, I would like to thank my Heavenly Father who provided me with insight and gave me the strength to complete this dissertation over the past few years.

I would like to thank my lovely wife, Rozanne Steyn, who stood by me during this time and who supported me up to the end. I would not have been able to complete this dissertation without her.

Special thanks to my family, who always seemed to be interested in the progress and motivated me to complete my work during difficult times.

Finally, I would like to thank prof. S. Gruner, my supervisor, for his time, insights, detailed review of my draft chapters and the suggestions he made. His input was very valuable.

LIST OF ABBREVIATIONS

ABP	Alternating Bit Protocol
ACP	Algebra of Communicating Processes
ADC	Analog to Digital Converter
ANTS	Autonomous Nano Technology Swarm
API	Application Programming Interface
ASP	Active Sensor Processes
BIP	Behaviour-Interaction-Priority
CBS	Calculus of Broadcasting Systems
CCS	Calculus of Communicating Systems
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
DFA	Deterministic Finite Automation
FDR	Failures-Divergences Refinement
FSM	Finite State Machine
FU	Functional Unit
GIS	Graphical Information Systems
GPS	Global Positioning System
GUI	Graphical User Interface
LRT	Long Running Transactions
MAC	Media Access Control
MEMS	Micro Electromechanical Systems
NASA	National Aeronautics and Space Administration
RF	Radio Frequency
SOS	Structural Operational Semantics
TCOZ	Timed Communicating Object Z
TPL	Task Parallel Library
VM	Virtual Machine
WSN	Wireless Sensor Network

TABLE OF CONTENTS

CHAPTER 1 Introduction	1
1.1 Problem statement	1
1.1.1 Context of the problem	1
1.1.2 Research gap	3
1.2 Research objective and questions	4
1.3 Hypothesis and approach	4
1.4 Research goals	5
1.5 Research contribution	5
1.6 Submitted paper	6
1.7 Previous publications	6
1.8 Overview of study	6
CHAPTER 2 Related work	7
2.1 Wireless Sensor Networks	7
2.1.1 Overview	7
2.1.2 Application areas	9
2.1.3 Evaluation metrics	12
2.1.4 WSN challenges	15
2.2 Communicating Sequential Processes	16
2.2.1 Process algebra	16
2.2.2 History	18
2.2.3 Application areas of CSP	19
2.2.4 Semantics	20
2.2.5 Extensions of CSP	22
2.2.6 Tools	27
2.3 Formal approaches to WSN modelling	29

2.3.1	WSN modelling using CSP	30
2.3.2	Other WSN modelling approaches	32
2.4	Chapter summary	33
CHAPTER 3 CSP theory		34
3.1	Traces in CSP	35
3.1.1	Calculating the traces of a process	35
3.2	Operators	37
3.2.1	Synchronous parallel	37
3.2.2	Alphabetised parallel	38
3.2.3	Interleaving	39
3.2.4	Generalised parallel	41
3.2.5	Optional parallel	43
3.3	Chapter summary	46
CHAPTER 4 Definition of a new optional parallel operator		47
4.1	Problem identification	47
4.2	Optional parallelism	50
4.2.1	Defining optional parallelism using classical CSP operators	51
4.2.2	Preparing system definitions for optional parallelism analysis	56
4.2.3	Generalisation of optional parallelism with classical CSP operators	61
4.3	Directional synchronisation for optional parallelism using channel artefacts	63
4.3.1	Directional notation	66
4.3.2	Broadcasting	68
4.3.3	Simplex	68
4.3.4	Half-duplex	69
4.3.5	Full-duplex	71
4.3.6	Notation and synchronisation summary	72
4.4	Other approaches considered	72
4.4.1	Using synchronisation event artefacts	72
4.4.2	Using stochastic CSP	75
CHAPTER 5 Software tools		77
5.1	Optional parallel to CSP definition generator	77

5.1.1	Requirements	78
5.1.2	System overview	79
5.1.3	Implementation	80
5.1.4	Test and validation	80
5.2	Automated trace verification	82
5.2.1	Requirements	83
5.2.2	System overview	84
5.2.3	Adjacency list notation	87
5.2.4	Theoretical trace generation	90
5.2.5	Simulated trace generation	99
5.3	Chapter summary	100
CHAPTER 6 Optional parallel test scenario description		101
6.1	Topology scenarios	101
6.1.1	Flat topology	103
6.1.2	Cluster	114
6.1.3	Chain	120
6.1.4	Tree	136
6.2	Chapter summary	143
CHAPTER 7 Testing optional parallelism in the traces domain		144
7.1	Test definition	145
7.1.1	Deadlock freedom	145
7.1.2	Trace refinement	146
7.1.3	Pass criteria	146
7.2	Limitations	147
7.2.1	Problems encountered	147
7.2.2	Solutions	148
7.2.3	Ignored metrics	148
7.3	Results	149
7.4	Discussion of results	152
7.4.1	Traces	152
7.4.2	Deadlock	154
7.4.3	Trace refinement	154

CHAPTER 8 Conclusion and outlook	155
APPENDIX A Trace results	172
APPENDIX B Topology Scenario Models	174
B.1 Flat Topology	174
B.1.1 Point-to-point	174
B.1.2 Fully Connected Mesh	177
B.2 Cluster Topology	184
B.2.1 Star	184
B.3 Chain Topology	190
B.3.1 3-Node Ring	190
B.3.2 4-Node Ring	194
B.3.3 4-Node Linear	199
B.4 Tree Topology	204
B.4.1 7-Node Tree	204

LIST OF FIGURES

3.1	Venn diagram for sets of first actions in generalised parallelism according to [1].	42
4.1	Basic WSN network graph with hyper-edge.	54
4.2	Basic network graph with hyper-edge and added channel artefacts.	54
4.3	Normal graph representation of the hypergraph of Figure 4.2.	58
4.4	Binary relationship conversion of Figure 4.3.	59
4.5	Cluster topology with processes P , Q , R and S	64
4.6	Broadcasting example showing before and after channel insertion.	69
4.7	Simplex example showing before and after channel insertion.	69
4.8	Half-duplex example showing before and after channel insertion.	70
4.9	Full-duplex example showing before and after channel insertion.	71
4.10	Example of 3-process hypergraph with additional synchronisation events added.	73
4.11	Event definitions for $N = 4$ and $M = 2$	74
5.1	Functional block diagram of OptoCSP.	79
5.2	Screen capture of the user interface of OptoCSP.	80
5.3	Functional block diagram of OpTrace.	85
5.4	Screen capture of the user interface of OpTrace.	87
5.5	Network graph and adjacency list for directional source P to destination Q . .	88
5.6	Network graph and adjacency list for hyper-edge with source P to destinations Q , R and S	89
5.7	Network graph and adjacency list for non-directional edge between P and Q .	89
5.8	Network graph and adjacency list for non-directional hyper-edge between P , Q , R and S	90
6.1	Point-to-point connection between processes P and Q	103
6.2	Broadcasting between processes P and Q	104

6.3	Bidirectional synchronisation between processes P and Q using a half-duplex synchronisation event.	105
6.4	Bidirectional synchronisation between processes P and Q using 2 simplex synchronisation events.	106
6.5	Fully connected processes P, Q, R and S	108
6.6	Broadcasting approaches between processes P, Q, R and S	109
6.7	Bidirectional synchronisation between processes P, Q, R and S using half-duplex synchronisation.	111
6.8	Bidirectional synchronisation between processes P, Q, R and S using 2 simplex synchronisation events.	112
6.9	Star topology with processes P, Q, R and S	115
6.10	Directional communication links for the star topology of Figure 6.9.	116
6.11	Bidirectional synchronisation between processes P, Q, R and S using half-duplex synchronisation.	117
6.12	Bidirectional synchronisation between processes P, Q, R and S using 2 simplex synchronisation events.	119
6.13	Ring topology with processes P, Q and R	121
6.14	Ring topology with processes P, Q and R	122
6.15	Bidirectional synchronisation between processes P, Q and R using half-duplex synchronisation.	123
6.16	Unidirectional synchronisation between processes P, Q and R using one simplex synchronisation event.	125
6.17	Ring topology with 4 node processes P, Q, R and S	126
6.18	Ring topology with processes P, Q, R and S	127
6.19	Bidirectional synchronisation between processes P, Q, R and S using half-duplex synchronisation.	128
6.20	Unidirectional synchronisation between processes P, Q, R and S using one simplex synchronisation event.	130
6.21	Linear topology with node processes P, Q, R and S	131
6.22	Linear topology with processes P, Q, R and S	132
6.23	Bidirectional synchronisation between processes P, Q, R and S using half-duplex synchronisation.	133

6.24 Unidirectional synchronisation between processes P , Q , R and S using one simplex synchronisation event.	135
6.25 Tree topology with processes P , Q , R , S , T , U and V	136
6.26 Tree topology with processes P , Q , R , S , T , U and V	137
6.27 Bidirectional synchronisation between processes P , Q , R , S , T , U and V using half-duplex synchronisation.	139
6.28 Unidirectional synchronisation between processes P , Q , R , S , T , U and V using one simplex synchronisation event.	141

LIST OF TABLES

3.1	Subset of the CSP notation, taken from [2].	35
4.1	Summary of synchronisation and channel definitions.	72
5.1	OptoCSP test results.	81
7.1	Result summary of topology tests.	152

CHAPTER 1

INTRODUCTION

1.1 PROBLEM STATEMENT

1.1.1 Context of the problem

Wireless Sensor Networks (WSN) are rapidly becoming a widely applicable technology in our everyday lives due to the advances in energy-efficiency, communication and processing power technologies as well as more cost-effective manufacturing solutions. WSNs are still prone to node and communication failures. The design and development of WSNs typically consist of emulation testing, simulation testing and laboratory test beds [3]. These testing methods are used to discover faults in the WSN software and communication protocols but cannot guarantee the absence of design faults. Proof-based formal methods are used to prove the absence of certain classes of faults in WSNs such as deadlock, livelock, bottlenecks [4] and safety properties [5]. Various process calculi have been defined for the reasoning of WSNs using formal models in aid of the formal verification thereof [6, 7, 8, 9, 10].

Communicating Sequential Processes (CSP) is a common process algebra used to formally specify the communication between processes. It was introduced more than 25 years ago [11] and has been applied in the fields of WSNs [12, 13, 14, 15], mission control systems, security protocols and transport control systems [16]. The semantics of CSP has been given in the operational, denotational and axiomatic forms by [2].

Classical CSP falls short to describe broadcasting, a communication construct used extensively in WSN communication protocols. A restriction of communicating processes under

parallelism in CSP is that all processes engaging in an event need to do so jointly [2]. This means that if one process engages in an event presented by its environment, it cannot proceed until all processes engage in the same event. This constraint often restricts the expressiveness of CSP on practical models like announcer/listener and reader/writer type problems as well as broadcasting protocols [17]. Existing operators of CSP like interleaving and a combination of interleaving and parallelism can be used to partially model the problem scenarios, but this is not sufficient, as its an approximation at best [17].

Optional parallelism was proposed in [17] to lift this restriction where it is not required for all processes under parallelism to jointly engage in events. With optional parallelism, a process may decide to opt-out of a communication event when its not ready to synchronise. The rest of the processes will then synchronise jointly on the event which allows the system to progress to its next state. An optional parallel operator will serve as a pragmatic addition to the CSP language to simplify models where total synchronisation is not practically viable. WSNs are common announcer/listener problems where an announcer may want to pass sensor information to all of its live listeners and bypass any node which is not ready to communicate or has run out of power. This should not block the network when information is to be traversed to the sink.

The semantics of optional parallelism can be derived from first principles or by defining optional parallelism using classical CSP operators. The former task is cumbersome and prone to errors while the latter approach is more viable because all the semantics of the classical CSP operators are already defined and formally verified by [2]. From [2] it is stated that any CSP operator can be deduced from a limited subset of CSP operators and that the current set of CSP operators are complete in the sense that *any* modelling concept can be modelled in CSP. From this, it is argued that optional parallelism can be defined in the context of classical CSP operators and inherit all the semantics in the operational domain.

This dissertation details the design of a translation of the definition of optional parallelism [17] into classical CSP operators which will enable optional parallelism to easily be used in existing CSP model-checkers. The optional parallel operator of [17] will be referred to as $OptPar$ and the translation presented in this dissertation as $OptPar^T$. $OptPar^T$ entails the addition of a channel modelling artefact in parallel composition with the existing process definitions which allows a communication event between the process and its environment to occur or not.

The processes are thus independent from each other with the channel artefacts orchestrating the synchronisation between them. If a process is not ready to engage in an event, its channel ignores the event, and the processes essentially opts out of synchronisation. Different combinations of the channel artefacts result in different directional synchronisation cases. Simplex synchronisation has the same behaviour as broadcasting. Duplex communication constructs are also defined which has the same behaviour as bidirectional synchronisation. Focus was given on the operational semantics of $OptPar$ in the development of $OptPar^T$. This resulted in a new optional parallel operator being defined with the same operational semantics of $OptPar$ as given in [17]. The operational semantics of $OptPar^T$ have been expanded to allow for directional synchronisation which better defines the notion of broadcasting and bidirectional communication links.

1.1.2 Research gap

Existing process algebras are often expanded [17] or used as a base to develop new process algebras [15] to add functionality to it. $OptPar$ intends to only be an additional operator to the existing set of classical CSP operators. It has its own operational semantics without modifying the existing operators' semantics to define its behaviour. Most of the new process algebras which are application specific like Timed Wireless Sensor Processes (Timed WSP) [15] need tool support to analyse system definitions using the new algebras. The model-checkers and proof tools are usually developed long after the new definitions and expansions of the process algebras, which makes it difficult for wide acceptance.

$OptPar$ is an addition to CSP which is not application specific. Optional parallelism can be used in various other applications such as announcer/listener and reader/writer type problems as well as broadcasting protocols [17]. The approach followed in this dissertation is to create a translation of $OptPar$ into existing classical CSP operators, forming $OptPar^T$. This allows optional parallelism to immediately be tool-supported in existing CSP model-checkers, unlike the other approaches followed.

1.2 RESEARCH OBJECTIVE AND QUESTIONS

The research objective of this dissertation is to derive a translation of $OptPar$ into classical CSP operators ($OptPar^T$) which will inherit the operational semantics of the existing CSP operators used in the translation. A trace relationship is to be found between $OptPar$ and $OptPar^T$ to test the correctness of the translation. $OptPar^T$ will then automatically be supported in existing CSP model-checkers because only existing classical CSP operators are used.

The research questions this dissertation will answer are:

- Is there a trace relationship between $OptPar$ and $OptPar^T$ of this dissertation?
- Can the optional parallel operator be used in WSN modelling scenarios?
- Can $OptPar^T$ be used in existing CSP model-checkers?

1.3 HYPOTHESIS AND APPROACH

The general hypothesis of the research to be conducted is:

“Optional parallelism can be defined using classical CSP operators to have the same behaviour in the traces domain.”

This will be done by deriving a solution based on the operational semantics of optional parallelism in the traces domain given in [17]. If a trace relationship exists between $OptPar$ and $OptPar^T$ for all possible scenarios, the hypothesis can be accepted. If there is one scenario where there is no trace relation, the hypothesis is rejected.

To test the hypothesis, the traces of systems defined using $OptPar$ have to be worked out, based on its step law and trace semantics given in [17]. If the traces are valid traces for the same system definitions using $OptPar^T$, a trace relationship exists and the hypothesis is accepted. If any counter example exists, the hypothesis is rejected. Due to the complexity and number of possible traces, a software tool is needed which implements the step law of $OptPar$ to work out the traces of system definitions using optional parallelism. Another software tool

is needed to convert a CSP system definition using $OptPar$ into a CSP system definition using $OptPar^T$. After the traces of a system using $OptPar$ has been worked out and the system converted into classical CSP operators ($OptPar^T$), the trace refinement relations can be tested using conventional CSP model-checkers.

1.4 RESEARCH GOALS

The research goals of this dissertation is to:

- define the behaviour of $OptPar$ in terms of classical CSP operators to form $OptPar^T$;
- test for trace refinement between $OptPar$ and $OptPar^T$;
- to have tool-support for $OptPar^T$ in existing CSP model-checkers; and
- to apply the notion of optional parallelism in CSP system definitions of WSNs.

1.5 RESEARCH CONTRIBUTION

This dissertation presents a solution to the notion of optional parallelism to be used in existing CSP model-checkers by defining $OptPar^T$ to have the same behaviour as $OptPar$ in the traces domain of CSP. Some differences in the trace semantics given in [17] have been documented. Software tools were developed to aid in the translation process and to perform scenario-based tests of CSP system definitions of WSNs containing the optional parallel operator.

$OptPar$ has theoretically been applied in [1] for an incremental Deterministic Finite Automaton (DFA) [18] minimisation algorithm to illustrate a possible application for the operator, but this application was made on the assumption that the initial definitions are correct and complete. The research presented in this dissertation applies optional parallelism in the field of WSNs to model the behaviour of communicating sensor nodes where intermittent node or communication failures could occur.

1.6 SUBMITTED PAPER

The following paper was submitted for the partial fulfilment for the degree Master of Engineering (Computer) and is still awaiting feedback on a possible publication:

- T. J. Steyn and S. Gruner, "A New CSP Operator for Wireless Sensor Networks," *EURASIP Journal on Wireless Communications and Networking*, Springer.

1.7 PREVIOUS PUBLICATIONS

Similar research has been done in the field of modelling systolic arrays using CSP by the author whereby a network communication graph is defined for a hexagonal systolic array and modelled in CSP. This model was used to prove deadlock freeness of the systolic array, in an attempt to generalise the proof to be used in model-checkers and theorem provers by examining one of many possible deadlock examples. The research methodology and contribution of the publication and this dissertation share commonalities in the field of CSP. The publication is:

- S. Gruner and T. J. Steyn, "Deadlock-freeness of hexagonal systolic arrays," *Inf. Process. Lett.*, pp. 539-543, 2010.

1.8 OVERVIEW OF STUDY

Chapter 2 gives an introduction of WSNs and its challenges. CSP is then introduced which also gives more detail on optional parallelism. The use of formal methods in WSN research is also given. Chapter 3 gives a brief introduction of the CSP notation used in this dissertation. Chapter 4 details the design of $OptPar^T$. The design and implementation of the software tools developed to test $OptPar^T$ is given in Chapter 5. Chapter 6 details the WSN test scenarios used to test $OptPar^T$. Chapter 7 gives the results and discussion of the test scenarios given in Chapter 6. Finally, Chapter 8 concludes this dissertation with a discussion and future work.

CHAPTER 2

RELATED WORK

The development of the operational semantics for optional parallelism will be based on the communication of WSNs. This facilitates the need to identify application areas of CSP in WSNs and for that an overview of WSNs is given first. After the definition of the notion of *process algebra*, CSP will be elaborated upon as it is the process algebra used in this dissertation. The topics discussed in this chapter are selected to cover all of the considerations and applications of the development of optional parallelism in CSP. Background on some of the topics such as WSNs and process algebras provide a foundation on which the extensions are identified and performed. Optional parallelism will then be introduced and the areas where it is applicable will be identified. Finally, the current situation on the tools available for CSP will be discussed as this is the end result of optional parallelism - to define the semantics thereof and encode it into a language understandable for a CSP model-checking tool.

2.1 WIRELESS SENSOR NETWORKS

2.1.1 Overview

WSNs are networks, typically consisting of small, energy-efficient and cost-efficient nodes, which relay *sensed* information to each other via wireless communication. The nodes are composed of sensor(s), data processing unit(s), communication electronics and an energy source. With these small nodes, a collaborative effort is made to sense some phenomenon. The sensors can be placed in a specific network topology or an ad-hoc fashion, either deployed inside a phenomenon or very close to it [19]. WSNs prefer broadcasting communication over

point-to-point connections due to the possible ad-hoc distribution of the sensor nodes. It is up to the communication protocol of the WSN to determine how the nodes communicate with each other and how errors are handled. Multi-hop communication reduces the energy usage of densely populated sensor networks because the transmitting ranges of the nodes can be kept to a minimum and the sensed data is propagated from node-to-node and eventually reaching the central node, which is referred to as the *sink* or *gateway*. This is in contrast with single-hop communication where a sensor node has to communicate with the sink directly.

In the past decade, WSNs have become a well-established research topic. This is due to the technological advances in integrated digital circuits, micro electronics and Micro Electromechanical Systems (MEMS). WSN design introduced various challenges, from energy efficiency to routing protocols with self-topological stabilisation. What further increased the feasibility of WSNs was the lower manufacturing costs and the advances in energy generation, storage and consumption. The energy consumed by WSNs were greatly reduced from the software side in the development of energy-efficient communication protocols [20, 21, 22, 23, 24] as well as the hardware side where the computing power increased with a lower energy requirement [25, 26, 27].

To design a wireless sensor network requires a detailed understanding of the capabilities and limitations of the hardware components. Furthermore, a detailed understanding of the networking technologies and distributed systems theory are needed. Each node needs to be designed with these limitations in mind from the ground up and to fully utilise all the available resources. One of the core challenges of WSN design is that the nodes need to be developed for its application area, as generic nodes would mostly not meet size, cost and energy criteria.

Wireless Sensor Actor Networks (WSAN) [28, 29] differ from normal WSNs with the addition of an actuator component. This allows the nodes to interact with their environment, based on the sensed phenomenon. This opens a new field of research because WSANs are now not only used for sensing parameters, but also to actuate on its environment, introducing a need for communication in the opposite direction as well.

2.1.2 Application areas

WSNs have been adopted to many fields in recent times. A common structure is shared among the applications where a distributed set of sensors periodically measures some phenomenon and then reports the measurements and computed results to a sink. Sensor nodes can be used for continuous sensing, event detection, event identification, location sensing and local control of actuators [19]. WSNs have been used for research and commercial applications like environment monitoring, military operations, space exploration, commercial products and structural monitoring. These *sense-and-send* [30] and *report-by-exception* [31] applications of WSNs are not limited to the mentioned fields and to cover them all would fall outside of the scope of this dissertation. Only a few application areas inspired by [31] will be discussed which covers most of the practical usage scenarios.

2.1.2.1 Sense-and-Send

A generic definition for a sense-and-send application is where the observer collects sensor readings from a set of points in an environment over a period of time in order to determine trends and dependencies. The data is collected from hundreds of nodes and analysed off-line. These applications often require a higher node density to minimise interpolation and extrapolation errors found in coarse deployment strategies. This also allows for long-term unattended operation to enable measurements at spacial and temporal scales. WSNs have the advantage that it can be deployed in ecological sensitive or challenging physical environments while also providing measured data in an unobtrusive manner by means of wireless communication. Research reported by [30] mapped sensor network measurements with Graphical Information Systems (GIS) for the exchange of geospatial data of the habitats of sea birds. This was done with minimal intrusion and with post-analysis, a typical procedure for environmental monitoring.

Environmental data collection applications, using tree-based routing topologies, can easily become energy bottlenecks because nodes with a lot of descendants transmit significantly more data than the leaf nodes. In the case of a full tree, the data to be transmitted increases exponentially. In many cases, the interval between transmissions can be in the order of minutes because of the slow variation in the environmental parameters typically measured.

In addition to the slow update rates, latency is also not a strict requirement because the data is usually analysed off-line. To increase energy efficiency, each communication event must be precisely scheduled, waking from a dormant state to transmit data at a precise schedule to avoid communication failures. When a node has run out of energy or a new node is inserted into the system, a new route should manually be configured.

The most important characteristics of environmental data collection applications is that the system have a long lifetime, precise communication events, low data rates and static topologies [31].

2.1.2.2 Report by exception

In security monitoring applications, the nodes are also placed in a fixed topology. The key difference in security monitoring applications is that data is not collected. Each node only sends data when a security violation is detected. This immediate, real-time communication events have a significant impact on the optimal network architecture. These types of WSN architectures are called *report-by-exception* networks [31].

With immediate and reliable communication being the primary requirement, an additional requirement is that it is confirmed that all the nodes are still present and in working condition. This status confirmation amongst the nodes requires less frequent communication events between them, at a frequency of approximately once per hour. The routing topology will be optimal if it has a linear topology, unlike the environmental data collection applications which have short, wide trees. This linear topology forms a Hamiltonian cycle of the network. A Hamiltonian cycle is a network path which visits each node exactly once. With this linear network topology, each node will only have one child, which distributes the energy consumption evenly in the network, unlike tree based topologies where there are energy consumption bottlenecks. The status confirmation between the nodes are needed due to the linear topologies not being fault-tolerant.

Energy is mostly spent on the low latency requirements of alarm signalling. With security monitoring applications, reducing the latency is more important than energy requirements and because these security violations occur infrequently, this design approach does not seem to be a problem. A common example of these networks are fire alarm systems, where the

signal needs to be routed, even if it means a high energy cost.

2.1.2.3 Report by detection

With node tracking applications, a tagged object's location can be detected and tracked through a region of space monitored by a WSN. This differs from conventional tracking applications like regular courier services, where the parcel is tracked at various checkpoints. While still useful, the object's current location cannot be determined, as it could have moved since its last checkpoint scan. This is however not practical for applications where the objects are not obliged to check-in at regular intervals to be tracked.

Node tracking is achieved by tagging the object with a sensor node, which will be tracked while moving through a field of other sensor nodes. The tagged object will be the phenomenon to be sensed by the WSN. This allows the object's current location to be known instead of where it was last scanned. This results in continual topology changes of the WSN as the object moves through the network. Another aspect to keep in mind is that the network should be able to detect new nodes as it enters the network, as well as current nodes exiting the network. This calls for routing protocols that are adaptable to continuous topology changes.

Two approaches to vehicle tracking and detection are given in [21]. The first approach is that the line of bearing of the vehicle is locally determined within the clusters before it is forwarded to the base station. Secondly, the opposite approach is followed where the raw data is sent to the base station where the location is determined.

2.1.2.4 Current and future areas

The National Aeronautics and Space Administration (NASA) is also one of the forerunners in WSN development. They are conducting research on sensor networks for planetary and solar system exploration. In their Mission to Planet Earth, they are developing WSNs for early warning systems for natural disasters as well as climate monitoring. These sensor networks are highly distributed and a high level of reliability is required, especially for the space missions, where the systems will need to operate in extremely harsh environments. Due to these requirements, the systems have a high complexity which results in a very large state space for testing. This makes it nearly impossible to test using conventional methods. A

suggestion made by [12] is to use tools which provide a high level of abstraction which can interpret system requirements and generate the software automatically from it. At the point of source code generation, the system will already be vigorously tested and qualified and the mathematical models verified to be correct.

2.1.3 Evaluation metrics

The metrics by which a WSN can be measured in terms of its performance are different for each WSN, depending on its application. The key evaluation metrics for WSNs at node or network level are lifetime, coverage, temporal accuracy, response time, effective sample rate, cost, ease of deployment and security [31]. These metrics are often interrelated, where a trade-off in one metric may result in an advantage in another. These metrics provide a multidimensional way to describe the capabilities of a WSN. This section will only cover these metrics briefly, as it is important to know of the existence of the metrics for future arguments. The detail however, is not required.

2.1.3.1 Lifetime

Lifetime is the most critical metric of WSNs. One of the main goals of WSNs is to leave the field nodes unattended for long periods of time. With energy as the primary limiting metric of WSNs, each node must be designed to manage its local supply of energy efficiently. The nodes can get an energy supply either from a battery (stored energy) or from the environment from solar cells or piezoelectric generators (harvested energy) [32]. Various energy sources for micro scale electronic devices are discussed in [33] and [34] amongst others. Energy efficient physical layer protocols are described by [21]. They add to their research by showing how to reduce energy consumption of non-ideal hardware through physical layer-aware algorithms and protocols.

2.1.3.2 Coverage

Coverage can be defined as how effective the sensor network monitors the field of interest and is thought of as a quality of service. Coverage is just as important as lifetime of a WSN. An important thing to keep in mind is that the coverage does not equal the range of a single WSN node. With multi-hop techniques, the coverage can be beyond the range of an

individual node. This is advantageous because it gives the WSN the ability to be deployed over a larger physical area. Although multi-hop networks can be infinitely large in theory, the networking protocols needed greatly increases the power consumption of the nodes which decreases network lifetime. Coverage is not always uniform for all nodes in the network. Heterogeneous sensor nodes could be used where the coverage of the nodes differs, which impacts the network topology directly. Scalability also becomes a factor with large networks, impacting lifetime and effective sample rates. More information on coverage is given in [35] and [36].

2.1.3.3 Temporal accuracy

In environmental and tracking applications, the samples from the nodes must be cross-correlated to provide meaningful and related data. The accuracy is proportional to the rate of propagation of the phenomenon being measured. Examples are temperature variations, which does not have a high rate of change, versus seismic monitoring in structural monitoring applications, which can have millisecond accuracy. Temporal accuracy is achieved by maintaining a global timebase for chronological samples. Continual time synchronisation requires higher energy expenditure as well as higher bandwidth, which are the main considerations for temporal accuracy.

2.1.3.4 Response time

In some applications, like reporting by detection, response time is critical. High-priority messages should be communicated immediately to their destinations. These messages have a low event frequency, but still require more energy than usual. This is because the system constantly needs to monitor its health by inter node communication. Energy is usually conserved by letting the nodes only activate their radio communication hardware for brief time periods, which will render detection systems ineffective. These systems must be continuously powered.

2.1.3.5 Effective sample rate

The effective sample rate is defined by the rate of which data can be sensed and communicated with the base station. This is a primary metric for data collection applications. It becomes

a bigger issue with multi-hop systems with a tree topology, where a node must handle all its leaf nodes' data, which can quickly become difficult to scale. Communication speeds (bit rates) as well as network size have a great impact on this metric.

2.1.3.6 Ease of deployment

One of the advantages WSNs has, is the ease of deployment. WSNs must be self-configurable and not require a skilled person for deployment. Automated deployment is especially used in areas which are topographically challenging to access. The network should be able to assess its own health and report constraint violations. Additionally, the system should also adapt to the changing environmental conditions and node relocations. With these requirements, WSNs can be deployed in inaccessible and environmentally harsh areas.

2.1.3.7 Cost

Some WSNs require hundreds of nodes to be deployed. Not only must these nodes be small, but their manufacturing costs kept to a minimum. Apart from deployment costs of the nodes, maintenance also plays a big role. Nodes constantly need replacement, especially in security monitoring applications. Self maintenance has an impact on energy requirements and effective sample rates.

2.1.3.8 Security

Information security is very important. Although some of the measured parameters seem useless when viewed out of context, it could be interpreted differently to identify a security hole. Power and light monitoring of a building could be intercepted to see when certain areas are occupied and could be used for a physical attack on the building's security. Data communication should also be authenticated, especially in the case of security monitoring systems, where a false alarm could be injected into the system. Encryption and authentication requires more power and network bandwidth due to the extra Central Processing Unit (CPU) power required for cryptographic computations and more data to be transferred due to encryption. Sample rate and lifetime is directly affected by security.

2.1.4 WSN challenges

In the habitat monitoring of sea birds work of [30], they used a single-hop and a multi-hop WSN configuration. Their network architecture consisted of a WSN, a common transit network and the base station. Small, battery-operated nodes, with application specific sensors were used. It was estimated that the lifetime of the nodes would be months to years, depending on the communication duty cycle. They had successful results with the single-hop network, but significantly less success with the multi-hop solution, attributed to poor resource management and complex routing protocols.

Coverage and network connectivity are common research topics for WSNs [35]. Coverage was briefly explained in §2.1.3.2. Connectivity is the ability of the sensor node to reach the data sink. Data can only be delivered if there is some network path from the node to the sink. The communication range of a node should not be confused with the sensing range as the sensing range is the area of observation, while the communication range is the area of inter-node communication of the sensed data.

2.1.4.1 Coverage

Coverage of a sensor node is usually depicted as a disk in 2-D or a sphere in 3-D. This makes a theoretical assumption that the node can successfully communicate with any node within the coverage area. This does not take obstacles like plants and trees or even furniture into consideration. Objects can block Radio Frequency (RF) signals or reflect them, inducing multipath communication characteristics. A study on the effect of objects on a network has been done by [37]. They simulated various object shapes inserted into a WSN simulation and determined the effect it had on routing protocols. Although the coverage problem was not addressed by them, their models could be used in coverage simulations.

2.1.4.2 Energy

One of the most important constraints a WSN has is energy constraints. Most sensor nodes depend on batteries as an energy source, but these batteries are mostly irreplaceable as the nodes are not always accessible. Several solutions have been developed to ameliorate the problem. A popular method is to place unused nodes into a low-power sleep mode, another

method which is also common is to tune the transmission range of a node to only reach a neighbouring node. The work of [19] suggests that multi-hop communication consumes less energy because of the transmission ranges being shorter and longer distances covered by multi-hop protocols. Hierarchical topologies allow cluster heads to aggregate data and reduce the information sent to the sink, relieving some of the burden on the nodes. Data gathering efficiency also reduces the power need. This is achieved by removing redundancy in the network so that the same phenomenon is not sensed by multiple nodes. Routing protocols opting for optimal node hops also conserve energy as the path with the lesser cost can be chosen for data transmission [35].

2.1.4.3 Reliability and fault tolerance

Some nodes may fail in the lifetime of the network due to power depletion, environmental interference and physical damage. Fault tolerance is the ability to sustain sensor network functionality without interruption due to node failures [38]. Some systems could also be designed to utilise physical damage as a form of sensing. Forest fire monitoring is an example of such form of sensing, where the nodes will be damaged as the fire-damage progresses, giving an indication of the affected area by tracking their last reported Global Positioning System (GPS) position. In contrast, the work of [39] focussed on the protection of the sensor nodes from the fire, using thermal insulation, which proves to be more cost friendly than to redeploy a whole WSN. With their sensor node protection research, the rate of fire spread can be easily detected for further research.

2.2 COMMUNICATING SEQUENTIAL PROCESSES

2.2.1 Process algebra

Algebra is a branch of mathematics related to the general properties of arithmetic. This branch is concerned with algebraic structures involving sets of elements with particular operations satisfying certain axioms. The goal is to derive general results, applicable to any example of the same algebraic structure, with the use of the set of defined axioms [40]. Elementary algebra follows the study of arithmetic which mostly consists of operations on sets of whole and rational numbers to solve first and second order equations. Relationships or

equations are summarised by using variables denoted by the letters of the alphabet to stand for unknown quantities whose values may be determined by solving the equations [41].

The behaviour of a system can be defined with processes and data. These systems can be anything from a vending machine to a complex concurrent system having many processing elements. Processes are commonly defined in terms of process graphs, but it is sometimes needed to perform algebraic calculations on it, hence the need for process algebra. Process algebras use a collection of operator symbols to specify and manipulate their process terms. An advantage of the symbolic definition of processes is that it is interpretable by a computer, automating process manipulations and verifications. Process algebras mainly consist of basic operators for the definition of finite processes, recursion to define infinite behaviour, and communication operators for concurrency. A further common notion is the definition of deadlock and a silent step. The operators are formally defined by structured operational semantics. The essence of process algebras is that equational reasoning can be performed on process terms to determine their behavioural relation with each other. This helps to check process graphs for equivalence. Extensions to process algebras are common to enhance expressibility, or for custom system behaviour. The structured operational semantics and equational logic are required for these extensions to be used in formal methods.

Process algebra provides a framework to formally reason about processes and data, with prominence given to concurrent processes, to detect system behaviour and properties. Systems can be verified for correct external output with the use of process algebra by expressing it as a collection of process terms with the use of the basic operators, recursion and concurrency. This is then manipulated by means of equational logic to prove the desired behaviour. [42]

Concurrency theory is the theory of parallel and distributed systems. Process algebra is regarded as an algebraic theory to formalise the notion of concurrent computation. Many frameworks for the description and modelling of reactive/concurrent systems exist of which process algebra is considered the dominant framework. Process algebra is suited for requirements specifications, design specifications and formal refinement proofs [43]. Process algebra became an underlying theory of all parallel and distributed systems which extends formal language and automata theory with the common notion of interaction [44].

2.2.2 History

After 12 years of research, the first publication of CSP was made by Hoare in 1985 [11] and its notation was extensively used in concurrency theory ever since. The development of CSP into a process algebra has gained wide acceptance after tools were developed to help model and analyse real life processes in CSP [2]. Together with its concurrent notational attributes, CSP has a collection of mathematical models and reasoning methods which helps with the understanding and use of the notation [2]. The primary interest for CSP is the interactions between processing elements on a communications level and is therefore defined relative to a given set of communications of processing elements. The basic processes of CSP are built from primitive processes such as SKIP and STOP and the communication primitives include sending and receiving data over a communication channel, internal and external choice distinction, parallel operators and sequential composition of processes. With these primitives, of which a subset is given in Chapter 3, CSP can be expanded to prove theorems as well as model the communication behaviour of complex distributed systems.

At the same time of development of CSP, other process algebras were also developed. One of the developments was Calculus of Communicating Systems (CCS) [45]. The actions of CCS model indivisible communications between exactly two processes. Formal language constructs include primitives for describing parallel composition, choice between actions and scope restriction. Message passing in CCS was inspired by [11] and was eventually handled similarly. Another process algebra, Algebra of Communicating Processes (ACP) [46], was developed around the same time as CSP and CCS. ACP is based on process algebra which defines alternative sequential and parallel composition. The foundation of the compositions was formed to analyse unguarded recursive equations [44]. With the addition of communications, the process algebra ACP was defined [44]. Within the development of ACP, the term "process algebra" was coined and is now used to group the families of mathematical theories of concurrency.

The three process algebras (CSP, CCS, and ACP) were developed at the same time and findings have been shared amongst researchers. CCS was the first algebra with a complete theory. CSP has distinguishable equational theory, which is different from the other two. ACP puts emphasis on the algebraic aspect in which there is an equational theory with a range of semantical models. ACP also has a more general communication scheme whereas CCS con-

siders communication which is combined with abstraction. CSP combines its communication with restriction, a useful attribute in the analysis of deadlock.

CSP makes use of multi-party communication which allows the broadcast of information to multiple receivers. This results that the distinction between input and output is vaguer than CCSs two-party communication. This non-distinguishable nature of CSPs input and output serves as a disadvantage for specification, but it is an advantage for verification. Constraint-based modelling methodology is only possible within a multi-party communication framework, which is a strong argument in favour of CSP as process algebra.

2.2.3 Application areas of CSP

CSP is the most widely used form of process algebra and has successfully been applied in areas of mission control systems, security protocols [47], transport control systems [48, 49, 50], multi-threaded programs [51] and WSNs [14, 52].

Systems are modelled in CSP to check the systems' properties before implementation. This gives a *formal mathematical* environment for verification of system properties. CSP has previously been used by [12] to model the Lights-Out Ground Operating System (LOGOS) from NASA. A post-implementation formal specification of LOGOS was done in CSP, which pointed out a number of anomalies and conflicts, not detected during their normal testing routines. These findings have gone undetected by system engineers, as was the case in [14], which already points out the importance of formal model-checking and verification.

Deadlock analysis is one of the most common reasons for the use formal verification techniques. It deals with detection, avoidance and prevention. A recent application of the deadlock analysis techniques from [53] was used by the author of this dissertation [54] to determine the deadlock properties of a hexagonal systolic array [55]. This work was based on the application of the techniques of [53] on a orthogonal systolic array presented in [53]. In both cases, the deadlock freedom was mathematically proved. This was manually accomplished from first principles, without the use of software verification tools.

2.2.4 Semantics

The study of semantics is the study of the meaning of languages. The semantics of computer programming languages entail the study of the formalisation of the practice of computer programming. A computer language consist of a syntax and semantics, the first describing the structure and the latter describing the meaning thereof [56]. The semantics of a programming language refers to the computational meaning of it, as opposed to its syntax. It is concerned with building mathematical models for understanding an reasoning about program behaviour. Semantics provide the relevant features of all possible executions of a language, ignoring the details which are irrelevant to the correctness of implementations. [57].

Semantics can be divided into two types. The first type is *static semantics* and the second is *dynamic semantics*. Static semantics are concerned with checking for well-formedness while dynamic semantics are concerned about the run-time behaviour of a program. In programming terms, static semantics are checks that are run at compile time while dynamic semantics are run-time based. In this dissertation, only dynamic semantics will be used.

Formal semantics allows for the translation from one domain to another formally defined domain and is detailed with the following sub groups of dynamic semantics:

- Operational Semantics;
- Denotational Semantics; and
- Axiomatic Semantics.

2.2.4.1 Operational semantics

Operational semantics offers a simple means of understanding how a program is intended to behave, and is of interest how the effect of a computation is produced [56]. The operational semantics of a computer language details how the syntax works in terms of rules or laws.

With operational semantics, the semantics of a program is specified as an abstract machine or transition system. All possible executions of the program are represented by the program's computations. Programs are interpreted as transition diagrams with visible and invisible

actions between states, and the main concern is how these states are modified during their step-by-step execution. The semantics of step-by-step execution are referred to as *Structural Operational Semantics (SOS)* [58]. The aim of SOS is to provide short and understandable semantic descriptions based on elementary mathematics [59]. *Natural semantics* [60] detail how the results of an execution are obtained *overall* with the relationship of the initial and final state of an execution. Operational semantics can be defined as the mathematical formalisation of some implementation strategy and provide simple guidelines for language implementation.

The operational semantics of CSP describe how the operators work in terms of their step laws, firing rules and trace clauses. This dissertation will focus on the step-by-step execution of optional parallelism in terms of its step laws defined in [17].

2.2.4.2 Denotational semantics

The framework of denotational semantics was first developed by Scott and Strachey [61]. With denotational semantics, the meaning of a program is modelled with mathematical objects which represents a function or a function-space. Only the effect of executing the functions are of concern and not how the effect is obtained. Denotational semantics describe what the syntax intends to achieve, but is not concerned with *how* it is achieved as in the case of operational semantics. The effect of each statement is given as an equation which describes the relation between the input and output state which results in a translation of the program to a well formalised mathematical domain [56].

2.2.4.3 Axiomatic semantics

Axiomatic semantics, primarily developed by Hoare in the 1960s [62], entail the expression of the properties of the effects of an executing construct as *assertions* in the form of a pair of predicates [56]. It provides a means of proving the properties of programs with the use of logical systems [56] together with pre- and post-conditions of program variables. A property is satisfied by a program if and only if the property can be proved from the axioms of the semantics. Axiomatic semantics can be used for the verification of program correctness using theorem provers.

2.2.5 Extensions of CSP

Classical CSP [2, 11] abstracts from optional parallelism [17], time-based communications [63], priority [64] and exceptions [65, 66], and it was extended to fit *specific* application areas. These extensions are not always expanded for all semantic models of CSP, which makes it difficult to be accepted as the standard. These extensions are more often abstract models, which are used to capture the extensional meaning of programs and to simplify the programs by reasoning in more abstract models. The advantage of these abstract models is that they introduce a theory of refinement which results that refinement checking naturally leads to specification and verification [65]. This subsection briefly describes some of the extensions of CSP and it should be noted that this is not an exhaustive list of the current extensions of CSP.

2.2.5.1 Optional parallelism

Optional parallelism (*OptPar*) was first defined by Roggenbach *et al* in [17] and is denoted by the operator \uparrow_X , where X denotes the synchronisation set just as in the generalised parallel operator of CSP. It extends classical CSP by adding a new operator for parallel composition of processes. The optional parallel operator is a hybrid operator incorporating interleaving and generalised parallelism. A restriction of communicating processes under parallelism in CSP, is that all processes engaging in an event need to do so jointly [2]. This means that if *one* process engages in an event presented by its environment, it cannot proceed until *all* processes engage in the same event. This often restricts the expressiveness of CSP on practical models like announcer/listener and reader/writer type problems as well as broadcasting protocols. Optional parallelism removes this restriction by defining the step laws for events where generalised parallelism is undefined. This is detailed in a Venn diagram in Figure 3.1 and §3.2.5.

OptPar has recently been used by [1] to conceptually specify an optimisation of an incremental DFA minimisation sequential algorithm as a concurrent system in CSP. *OptPar* was successfully used to formally specify the algorithm as set of concurrent functions and the use of formal model-checkers would have been useful to verify the correctness of it. Further research and investigation beyond the scope of this dissertation is needed to define the work

of [1] in a CSP model-checker for formal verification.

The operational semantics of optional parallelism was defined in [17] in terms of the step laws (given in §3.2.5) and firing rules. Additional theorems were given of which the outline of the proofs exist, but only one is found in the literature [17]. Details on failures and divergences have been sketched but not detailed.

Some differences between $OptPar$ and $OptPar^T$ exist. First, with $OptPar$ there is no trace law for two processes under optional parallelism where one opts out of synchronisation on an event and then the other opts out of that same event directly thereafter. With this in mind, consider the processes $P = Q = comm \rightarrow STOP$. $Traces(P \underset{comm}{\uparrow} Q)$ does not contain the trace $\langle comm, comm \rangle$ as interleaving would and it is also argued in [17] that there is no refinement relationship between interleaving and optional parallelism due to this counter example. The definition of $OptPar^T$ given in Chapter 4 allows the trace $\langle comm, comm \rangle$ as it is believed that process P may opt out of synchronisation of $comm$ and be ready to synchronise just after Q performed its $comm$ event, even if Q will not be ready the second time. This has the effect of interleaving when a process is not ready to synchronise.

Secondly, there is also no trace law for $OptPar$ in [17] for the clause of Equation 3.37 where $a \neq b \wedge a, b \in X$. This means that deadlock will occur if two processes are ready to communicate different events in the synchronisation set. $OptPar^T$ treats this scenario as interleaving.

2.2.5.2 Timed CSP

Timed CSP extends the CSP language of Hoare [11] to include timing concepts. It was first published by Reed and Roscoe [67] after which various denotational semantic models have been published by the authors [67, 68]. Davies and Schneider provided an introduction on these models in their work of [69]. The language and models have been refined to have a more abstract view of time and these changes was noted in [70]. The timed model of [71] is based on the notes of [70]. Classical CSP processes describes behaviours between processes and the most basic behaviour is a sequence of events, not taking time into consideration. Timed CSP achieves the same, but with the addition of a *time* when these events are performed or refused. Timed CSP proved to be more powerful because a Timed CSP process can be syntactically

transformed to an untimed CSP process while preserving much of the information. With the appropriate conditions, properties of the Timed CSP process can be formally defined and verified by the study of its untimed transform. This will also be the approach for the definition of optional parallelism, where processes can be translated to classical CSP. For a more recent detailed history and working of Timed CSP, see [72].

2.2.5.3 Timed WSP

Timed WSP was introduced by Liu *et al* [15] and is based on Timed CSP and the Calculus of Broadcasting Systems # (CBS#) [6], an extension of CBS given by [73]. In the language of Timed WSP, the notions of time and broadcasting are used to describe the behaviour of WSN nodes. They used Timed WSP to model contention-based WSNs which are abstracted from existing Media Access Control (MAC) protocols. Liu *et al* introduced two new process prefixes to define transmitting and sending processes which are used in broadcasting definitions. The associated operational semantics was given in [15] and tool support was still a topic of future work in their research. Liu *et al* argue that their Timed WSP solution is suitable for formal WSN specifications to describe broadcasting and collision avoidance. The notation of their transmitting and receiving processes are given as $\otimes m \rightarrow P$ and $\odot m \rightarrow P$ respectively.

The optional parallel translation of this dissertation also details the notion of directional communication in terms of transmitting and receiving channel artefacts which are detailed in §4.3.1. The approach of this dissertation differs from the approach of [15] as existing CSP operators are used and therefore the solution is tool supported by design.

2.2.5.4 Compensating CSP

Compensating CSP (cCSP), introduced by Butler *et al* [74], is an extension of CSP to model Long Running Transactions (LRT) which is an error recovery mechanism for these transactions. It was inspired by transaction processing features commonly found in database and on-line banking mechanisms. Processes in cCSP are still modelled in terms of atomic events as in classical CSP and the language supports standard CSP constructs of sequencing, choice and parallel composition. Additional compensating operators were added to support the long running transactions and the failure occurrences of it. Processes are categorised into standard and compensable processes where the compensable processes have an attached compensation

definition which is used when compensation is needed.

The operational semantics of cCSP was further detailed in [75] together with an outline of the encoding thereof in Prolog. The operational semantics of [75] provides the basis for a prototype model-checker for cCSP as well as a foundation to implement it in a language with compensations. The work of [76] defines a technique which allows LRTs to be modelled with cCSP and to be translated into the Promela language to be analysed by the SPIN tool. Their work shows promising results but is still only a work in progress.

A notational difference in process definitions is where P and Q represent normal processes, PP and QQ represents compensable processes. A compensable process is a normal process with compensation actions attached to it. The process consist of a forward behaviour and a compensation behaviour. The compensation behaviour will be executed in the case of a fault or exception to compensate the forward behaviour. Both the forward and compensable behaviour are standard processes. A process definition with its compensation is defined as $P \div Q$ with P as the forward behaviour and Q as the associated compensation. Q is designed to compensate for the effect of P and may run long after P has completed. For more information on the operational semantics of cCSP, see [57] for details. The syntax of cCSP is summarised as follows.

Standard Processes:		Compensable Processes:	
$P, Q ::= A$	(atomic action)	$PP, QQ ::= P \div Q$	(compensation pair)
$P ; Q$	(sequential composition)	$PP ; QQ$	
$P \square Q$	(choice)	$PP \square QQ$	
$P \parallel Q$	(parallel composition)	$PP \parallel QQ$	
$SKIP$	(normal termination)	$SKIPP$	
$THROW$	(throw an interrupt)	$THROWW$	
$YIELD$	(yield to an interrupt)	$YIELDD$	
$P \triangleright Q$	(interrupt handler)		
$[PP]$	(transaction block)		

2.2.5.5 Termination in CSP

The work of Howells and d’Inverno [77] suggests that the failure-divergence semantic model of CSP has an incomplete treatment of successful process termination. They focus their work particularly on parallel termination which permitted unnatural definitions of processes where

events can occur after the termination event \checkmark . An analysis of the existing CSP synchronous and asynchronous parallel operators indicated that none of them had asynchronous termination semantics. Solutions to this problem was suggested by Hoare [11], Tej and Wolff [78] and Roscoe [2]. Howells and d’Inverno presented another solution and called it CSP_T and they argue that it is closer to the original semantic model which provides greater flexibility over the type of parallel termination semantics of CSP.

Howells and d’Inverno introduced a new CSP axiom which captures the notion of successful termination and modified the semantics of CSP to do so. They also showed that the non-parallel processes still satisfy the termination axiom. Finally, they introduced replacement parallel operators for CSP which satisfies the new termination axiom. The three parallel operators they added are:

- \parallel_{\emptyset} - **Race Parallel** - If P or Q terminates asynchronously, $P \parallel_{\emptyset} Q$ will terminate. Race parallel fails to terminate if and only if both P and Q fails to terminate.
- \parallel_{\emptyset} - **Asynchronous Parallel** - P and Q both needs to terminate for $P \parallel_{\emptyset} Q$ to terminate. Asynchronous parallel fails to terminate if either P or Q fails to terminate or both P and Q fails to terminate.
- \parallel_{Δ} - **Synchronous Parallel** - The termination event \checkmark is included in the synchronisation set and will therefore allow synchronised termination on \checkmark .

They conclude that their addition of these new parallel operators provides two advantages over the solutions by Roscoe [2]. First, it is simple in the sense that it requires fewer modifications to the existing CSP semantics and secondly because it provides a greater flexibility in the choice of termination semantics for the parallel operators provided.

Termination of optional parallelism is defined such that both processes in a binary optional parallel relation should terminate jointly for optional parallelism to terminate successfully. In the case where one process terminates before the other, deadlock occurs with the observation of the *STOP* trace. This behaviour is similar to the asynchronous parallel operator defined above.

2.2.6 Tools

Formal methods provide a means for exhaustive system analysis, as opposed to simulation where scenarios are often created which do not cover all possible system states. One of the challenges of using formal methods is to create and establish techniques which the industry can use. This is measured in terms of people being able to utilise the methods and if the methods deliver useful results for realistic systems. Formal verification of systems with real-world application requires the help of computers. The type of computer aided support comes in the form of model-checkers and theorem provers, or the combination thereof.

Model-checkers are specialised to one particular formal method and are based on an exhaustive search of a finite state space. These tools do not require user intervention. A property is defined at the start by the user and the tool shows the property or fails. Model-checking provides support for positive system specification verification, i.e. indicating if a specification is successfully implemented, as well as debugging information like stack traces where failures occur. This is more useful during early development since it helps to reduce errors. One of the disadvantages of model-checkers is that they can only handle systems of finite size, making it difficult to prove that systems will be error free for arbitrary sizes.

Theorem provers are capable of verification where a system's specification can be verified for all instances of its parameters. This comes with a price because human interaction is needed together with the computer aided tools. When proofs are not possible, debugging information is often not included which makes problem finding difficult for the user. General purpose theorem provers are based on powerful meta-logic like higher-order logic or type theory into which the formal method is embedded [79]. Theorem provers assist only in the construction of proofs and the main work is left to the user.

Theorem provers have been suggested to complement the well-established technique of model-checking [79]. The advantages of the two groups of formal analysis tools are complimentary. Model-checkers are fully automated and highly specialised where theorem provers are user-driven and provide a flexible and versatile formal proof environment. The combination of model-checkers and theorem provers provides powerful formal proof and development environments for processes.

CSP-Prover [80] is a combination of a model-checker and a theorem proving tool, which specifically targets proofs for infinite state systems, which may also include infinite non-determinism. CSP-Prover is based on the theorem prover Isabelle [81] which provides a deep encoding of CSP and supports various analysis models including the failures model and the traces model of CSP. Practical applications of CSP-Prover have been demonstrated with success where an electronic payment system was modelled after it was formalised in CSP. Tool support was needed [80] to prove the system as deadlock free because of the complexity of the model of the electronic payment system. This was also found necessary for the verification of the system specifications in [43]. These proofs were based on the fact that CSP processes can be deadlock-free by design and that CSP's stable-failures refinement is deadlock-preserving [82]. The power of CSP-Prover was also verified by successfully analysing the dining philosophers problem which is a classical mutual exclusion problem and tailored to be a benchmark of refinement proofs [80].

A high-level formal modelling tool was developed, Requirements to Design to Code (R2D2C), based on CSP [12]. It was designed to aid the system specification and verification of WSN applications. It has been shown that the tool allows a formal model to automatically be transformed to source code. This allows the developer to define the system formally using CSP and using the R2D2C tool to generate the source code. This currently only supports natural language or semi-formal language as input, a conversion to a subset of CSP, called EzyCSP, and Java source code as output. The R2D2C tool can be applied in other application fields as well, although it was specifically tailored for WSN applications.

The model-checker Failure-Divergence Refinement (FDR¹) [83] is regarded as the standard proof tool for CSP and provides refinement proofs as well as deadlock and livelock analysis. A limitation of the earlier proof tools such as FDR1 is that only finite state systems could be analysed and only concrete data types could be used. These proof tools suffered in practical applications where state explosion problems were introduced [84]. FDR2 was developed as an improvement over FDR1 in the sense that it supported operators outside of the CSP core and hence supported other languages, improved handling of multi-way synchronisation, provisions for much more power language for data types and expressions, a potential for lazy exploration of systems and the ability to build up systems gradually. FDR2 is only available for the Linux operating system in both x86 (32-bit) and x64 (64-bit) formats. With the advances

¹<http://www.fsel.com/>

in computer technology, FDR2 has been adapted to utilise the new processing power made available by multi-core processors. Parallel FDR2 [85] is very powerful in high fidelity models where it can produce a lot more detail than conventional FDR1. The latest version, FDR3² [86], is a complete rewrite of FDR2 and has a significant number of improvements. The most prominent enhancements of FDR3 are a new multi-core refinement-checking algorithm, able to deliver a near linear speed increase with an increase in the number of cores, and a new algorithm for efficient internal CSP representations. Another non-functional enhancement was the improved ease of use of FDR3 with a powerful Graphical User Interface (GUI). It is available for Linux (x86 and x64), Microsoft Windows (x64), and MacOS (x64) and includes an Application Programming Interface (API) for C++, Java and Python, detailed in its manual [87]. Experimental results [86] have shown that FDR3 is faster and less memory intensive for a single thread versus FDR2.

ProB³ [88] is an animation and model-checking tool for the B method. It contains a temporal and a state-based model-checker to be used to detect errors in B specifications. Later on, ProB was extended to support CSP as well and is called ProCSP. ProB is used within Siemens, Alstom and several other companies for complex data validation. Commercial support is given by a company called FormalMind⁴. ProCSP has a user friendly GUI and multi-platform support, which makes it a viable choice for CSP system analysis. At the time of writing this dissertation, only ProCSP was available as an easy CSP tool to use as FDR3 was in its final stage of development.

2.3 FORMAL APPROACHES TO WSN MODELLING

The nature of WSNs after deployment often does not allow access to the nodes in the case of system failures. It is therefore essential to simulate and analyse the network design beforehand to minimise the risk. Model-checking is one of the design verification approaches taken. Protocol analysis and system properties such as deadlock, livelock and safety are some of the many verification parameters model-checking enables.

²<https://www.cs.ox.ac.uk/projects/fdr/>

³<http://www.stups.uni-duesseldorf.de/ProB/>

⁴<http://www.formalmind.com/>

2.3.1 WSN modelling using CSP

A method to check ad-hoc wireless networks for self-stabilisation is given in [14]. CSP and FDR were used as the process algebra and model-checking tool. Tactical networks were targeted first by [14] before applying the findings to more general self-configuring networks. The work was inspired by the ARPANET bug [89], where a corrupt topology update message caused a livelock which required all the routers in the network to be reprogrammed with a patch and then manually rebooted. This could have been avoided if the network state combinations were formally modelled and verified. With the application of CSP modelling on a tactical internet, failure state properties of the network were detected by [14] from the CSP_M models developed, a task previously done by intense manual testing. This automated capability to find protocol flaws was unknown at the time. In [14], system models were limited to a small finite number of principles. A verification approach of inductive reasoning was used, which in CSP and FDR is typically based on data independent induction [13, 90]. The models were initially constructed for the most simple form of a sensor node and then gradually refined until it contained most of the network's physical protocol behaviour. With these rich models, FDR derived scenarios where the network could perpetually reconfigure itself. The work of [90] tried to address the problem of scalability by combining data independence with induction, two of the best known approaches. A complex road network example was modelled in CSP and verified as deadlock free with FDR. High fidelity models in FDR were still found by [14] to suffer from state space explosion [84], even with the state reduction operators provided by FDR [2]. A more abstract approach was taken by [90] in the protocol models by means of induction with successful routing analysis by FDR.

A simple wireless sensor network was modelled by [12] and it was used to develop a custom tool for automatic model generation and verification. CSP was used to create the model and FDR was used as the model-checking tool. A novel construct of dividing the WSN into regions with the regions placed collectively into a synchronous parallel composition was used, which reduced the model states. A method to speed up WSN application development was also suggested, albeit sketchy. The Autonomous Nano Technology Swarm (ANTS) [91], a space based WSN, which still in its concept development phase at NASA was modelled in CSP [92]. It was found that CSP was particularly powerful in the specification of communication protocols as well as the analysis of race conditions, a phenomenon which is very common in

swarm systems.

A new notation was developed by [93], called Active Sensor Processes (ASP). This is based on Timed-CSP [94], Timed Communicating Object Z (TCOZ) [95] and specific hybrid-broadcasting extensions. It is argued that pure CSP, CCS and π -calculus are not suited for WSN modelling because they rely on on a single communication mechanism, not suitable for inter-sensor communication. Credit is given to CSP and CCS by stating that the process algebras are well suited for parallel composition *within* sensor nodes. On the other hand, it is stated in [96] that CSP is naturally suitable to describe distributed measurement systems, using independent communicating sensors. The π -calculus gets its credit from the ability to define dynamic reconfigurable networks by allowing channel names or the residing location of a process to be changed. It was also shown that previously unknown design flaws can be detected with existing system analysis methodologies. It is, however, argued that systematic sensor network verification in general may need verification techniques beyond the capabilities of existing methodologies. ASP was used as the specification language, together with UPPAAL [97], a verification tool for real-time systems. The limitations of UPPAAL were quickly reached, where it took a lot of time to analyse a mere 6 sensor nodes. As like [12], [93] are also developing a tool for automatic code generation from ASP to a native language. This is, however not yet mature enough to be used. Other work closely related to ASP is SensorML [98].

Another sensor network was formally defined using funclets [96]. These are small sensors constructed from small functional elements. A funclet is described as a special CSP process which is started by an external triggering event and with internal events which are not allowed to handle these events. The funclet notation, which is based on CSP syntax with simplifications for practical purposes, was used. The funclet notation was specified by an earlier paper, [99], which contained the syntax definition and semantics. Deadlock freedom could be proved, but it could not be determined if it was livelock free.

A deadlock prevention approach was taken by [96] for their sensor network network design. A Virtual Machine (VM) based WSN was designed to structurally guarantee deadlock freedom. This was done by defining the system as a basic client-server system with a circuit-free client-server digraph to ensure deadlock freedom [100]. CSP was used as a specification language for the sensor networking applications together with the CSP digraph definition of [100].

2.3.2 Other WSN modelling approaches

Various other approaches to WSN modelling have been investigated. The work of [101] uses Behaviour-Interaction-Priority (BIP) [102] to model and verify WSNs. Their work focussed on a methodology to build the model of a node from its nesC [103] application software together with its TinyOS⁵ operating system component models into BIP component models. A network model is formed with the composition of the BIP component models using BIP connectors [104] implementing different types of radio channels. Their approach is unique because they model real world sensor node application software together with its underlying operating system which results in non-deterministic and fully characterised WSN models. This enables the systems to be model-checked as they have a well-defined notion of state [101]. They had positive experimental results in different systems tested due to the exhausted state space exploration functionality of the *Observer* component of BIP. Various traces and error states could be detected, based on different system input parameters.

Event Calculus [105], an event-based formal language, was used by [4] and [106] to also perform static analysis of WSNs. The work of [4] focussed on correctness and structural specifications of WSNs and to form dependability metrics such as lifetime, connection resiliency and coverage. In [106], the authors apply Event Calculus to analyse the performance of WSNs in terms of the power consumption and lifetime of the individual nodes. The approach of [4] used an Event Calculus reasoner, called the Discrete Event Calculus (DEC) Reasoner, as the mathematical engine of their custom Java-based software to analyse WSN systems. This approach is similar to the work of this dissertation where the process of checking WSN designs are automated as far as possible with the development of a front-end tool which has a mathematical back-end to perform the systems analysis based on the chosen formal language. The Event Calculus used by [4] allows the open issue of formal specifications where the specification changes if the WSN changes to be addressed. They devised a general specification for the correctness of WSN properties which is valid for any WSN together with a structural specification which is unique for every WSN. Their Java-based software tool automates the structural specification of the WSN. Positive results were obtained from their case studies and they could show the correctness properties of the WSNs analysed. The main difference between the work of [4] and this dissertation is that model checking is possible with the use of

⁵www.tinyos.net

CSP, something still a topic of future work for [4]. Whereas the optional parallelism approach of this dissertation is designed for *static* networks, which cannot change their graph structure during their lifetime, *dynamic* networks, which can change their graph structure, are *not* in the scope of this dissertation. Latest work about the process-algebraic formalisation of dynamic networks can be found, for example, in [107].

2.4 CHAPTER SUMMARY

This chapter discussed many concepts and a plethora of research topics exist within these fields of study. This can be narrowed down to the formulation of an applicable concept in the field of formal methods, especially CSP, a well established and extensively researched process algebra. To argue the applicability of the new concept of optional parallelism in CSP, an overview of WSNs was given to sketch an application area in which CSP could be used to create mathematical models of real-world WSNs. This creates a foundation to test and develop the operational semantics of optional parallelism. The challenges presented by WSNs indicate a need for formal verification via model-checking techniques and WSNs seem to be an ideal application to test and apply optional parallelism in CSP models. There are many simulation tools and formal verification methods used currently, and with great success. It is argued that these formal models could be simplified with the use of optional parallelism, where faulty communication channels and WSN nodes as well as broadcasting are generically incorporated in the operational semantics of optional parallelism.

CHAPTER 3

CSP THEORY

CSP has a set of operators and laws to define and model communicating processes. Set theory and general algebra are also used for system definition, system analysis and theorem proving. This chapter only gives an overview of the CSP operators used in this dissertation. The information given in this chapter is put here as a reference to the reader which is not so familiar with the operational workings of CSP. The details are extracted from Roscoe's book *The Theory and Practice of Concurrency* [2]. The full set of classical CSP operators and their semantics in the different domains of CSP can be found in [11] and [2]. Focus will be given to the concurrency operators of CSP where independent processes interact with each other when necessary. Table 3.1 gives the CSP notation used in this dissertation.

Notation	Description
$a \rightarrow P$	Event a and then behave as process P
$x?A \rightarrow P(x)$	Choice of x from set A then $P(x)$
$P \setminus X$	Events in X hidden by P
$P \triangleleft b \triangleright Q$	If b is true then P , else Q
$P \parallel Q$	P in synchronously parallel with Q
$P \parallel_X Q$	P in parallel with Q with synchronisation set X
$P \parallel_X \parallel_Y Q$	P in alphabetised parallel with Q with synchronisation set $X \cap Y$
$P \parallel\parallel Q$	Process P interleaves process Q
$P \uparrow_X Q$	Process P in optional parallel with process Q with synchronisation set X . <i>OptPar</i> [17]

Notation	Description
$P \underset{X}{\parallel} Q$	Process P in translated optional parallel with process Q with synchronisation set X . $OptPar^T$
$P \square Q$	External choice between processes P and Q
$P \sqcap Q$	Internal choice between processes P and Q

Table 3.1: Subset of the CSP notation, taken from [2].

3.1 TRACES IN CSP

The trace of a process is a recording of the events which the process communicates. These events are the observable sequences of communication from the perspective of the environment, and any internal events of a process will be hidden from such observation. Traces may be finite or infinite. Finite traces are found either when the observation was terminated, or because the process and the environment have reached a point where they do not agree on any event. Finite traces provide a means of what can effectively be observed. Infinite traces are observed when the system is observed forever and infinitely many events are recorded, which is needed to capture fairness. The recording of the observed traces emphasises on the sequence of events and not the time at which an event has occurred.

An untimed CSP process or system can be characterised by the set of all possible traces that can be emitted. Recording only the finite traces is sufficient in the majority of cases because in an infinite trace, all its finite prefixes are finite traces.

3.1.1 Calculating the traces of a process

For any process P , $traces(P)$ will always have the following properties:

- $traces(P)$ is non-empty and will always contain the empty trace $\langle \rangle$;
- $traces(P)$ is prefix-closed, if $s \hat{=} t$ is a trace then at some earlier stage during the recording of the traces, the trace was s .

For any process P , $traces(P)$ is defined to be the set of all its finite traces. These traces are members of Σ^* , the set of finite sequences of events¹. The following examples from [2](p. 36) illustrate this:

- $traces(STOP) = \{\langle \rangle\}$ - The empty trace is observed if no event can be performed;
- $traces(a \rightarrow b \rightarrow STOP) = \{\langle \rangle, \langle a \rangle, \langle a, b \rangle\}$ - this process may have communicated nothing yet, performed an a only, or an a and ab . Note that b cannot be observed before a is observed;
- $traces((a \rightarrow STOP) \square (b \rightarrow STOP)) = \{\langle \rangle, \langle a \rangle, \langle b \rangle\}$ - here there is a choice of first event, so there is more than one trace. Note that ab can not be observed as in the previous example as each side stops execution after its event was observed;

The traces of a process enable its CSP notation to be given meaning or semantics as well as to describe the behaviour thereof. The set of all non-empty, prefix closed subsets of Σ^* is called the traces model, written as \mathcal{T} , which is the set of all possible traces of a process. The traces model is the simplest of CSP models and will be the model used in this dissertation.

The following simple trace rules for CSP's basic operators exist [2](p. 37). The \wedge operator denotes the concatenation of traces. More advanced operator's trace rules are covered in the next subsection.

- $traces(STOP) = \{\langle \rangle\}$;
- $traces(a \rightarrow P) = \{\langle \rangle\} \cup \{\langle a \rangle \wedge s \mid s \in traces(P)\}$ - this process has either done nothing, or its first event was a followed by a trace of P ;
- $traces(?x : A \rightarrow P) = \{\langle \rangle\} \cup \{\langle a \rangle \wedge s \mid a \in A \wedge s \in traces(P[a/x])\}$ - this is similar except that the initial event is now chosen from the set A and the subsequent behaviour depends on which is picked: $P[a/x]$ means the substitution of the value a for all free occurrences of the identifier x ;
- $traces(c?x : A \rightarrow P) = \{\langle \rangle\} \cup \{\langle c.a \rangle \wedge s \mid a \in A \wedge s \in traces(P[a/x])\}$ - the same except for the use of the channel name, c ;

¹Termination \checkmark is not mentioned here as the scenarios of this dissertation do not deal with termination.

- $traces(P \square Q) = traces(P) \cup traces(Q)$ - this process offers the traces of P and then those of Q or *vice versa*;
- $traces(P \sqcap Q) = traces(P) \cup traces(Q)$ - since this process can behave like either P or Q , its traces are those of P and those of Q ; ²
- $traces(\sqcap S) = \bigcup \{traces(P) \mid P \in S\}$ - for any non-empty set S of processes;
- $traces(P \lt b \gt Q) = traces(P)$ if b evaluates to *true*; and $traces(Q)$ if b evaluates to *false*.

3.2 OPERATORS

This section gives more details of the CSP concurrency operators used in this dissertation from Roscoe's book *The Theory and Practice of Concurrency* [2]. The step laws, general algebraic laws of associativity, symmetry and distributivity along with the trace laws are given.

3.2.1 Synchronous parallel

Synchronous parallelism is the simplest form of parallelism in CSP, but also the most restrictive form of parallelism. The synchronous parallel operator requires that processes agree on *all* possible events that can occur. It is written as $P \parallel Q$. More information on the synchronous parallel operator can be found in §2.1 of [2].

3.2.1.1 Laws

Step law

$$?x : A \rightarrow P \parallel ?x : B \rightarrow Q = ?x : A \cap B \rightarrow (P \parallel Q) \quad (3.1)$$

Symmetry

$$P \parallel Q = Q \parallel P \quad (3.2)$$

²Thus, the traces cannot show whether a choice was made internally (\sqcap) or externally (\square).

Associativity

$$(P \parallel Q) \parallel R = P \parallel (Q \parallel R) \quad (3.3)$$

Distributivity

$$P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R) \quad (3.4)$$

3.2.1.2 Trace semantics

The traces are simple to compute. It is the intersection of the individual traces of P and the traces of Q .

$$\text{traces}(P \parallel Q) = (\text{traces}(P)) \cap (\text{traces}(Q)) \quad (3.5)$$

3.2.2 Alphabetised parallel

When using the synchronised parallel operator, the more processes added, the more processes need to agree on every event. This behaviour is not always wanted and a more general version of parallelism exists, called alphabetised parallelism. This allows events which are in P and not in Q to be executed without synchronisation and only events which P and Q share to be synchronised. The same condition holds for events in Q which are not in P . This is stated more formally from §2.2 of [2] below.

If X and Y are subsets of Σ , $P \parallel_X \parallel_Y Q$ is the combination where P is allowed to communicate in the set X , called its *alphabet*, Q is allowed to communicate in its alphabet Y , and they must agree on events in the intersection $X \cap Y$. Thus $P \parallel_{\Sigma} \parallel_{\Sigma} Q = P \parallel Q$.

The processes P' and Q' denote a transition to a next state of their respective process definitions P and Q . This notation will be used throughout this chapter.

3.2.2.1 Laws

Step law

$$P \parallel_X \parallel_Y Q = ?x : C \rightarrow (P' \ \langle x \in X \rangle \ P \parallel_X \parallel_Y \ Q' \ \langle x \in Y \rangle \ Q) \quad (3.6)$$

Symmetry

$$P \ X \parallel_Y \ Q = Q \ Y \parallel_X \ P \quad (3.7)$$

Associativity

$$(P \ X \parallel_Y \ Q) \ X \cup Y \parallel_Z \ R = P \ X \parallel_{Y \cup Z} \ (Q \ Y \parallel_Z \ R) \quad (3.8)$$

Distributivity

$$P \ X \parallel_Y \ (Q \sqcap R) = (P \ X \parallel_Y \ Q) \sqcap (P \ X \parallel_Y \ R) \quad (3.9)$$

Indexed notation

$$\parallel_{i=1}^n (P_i, X_i) = P_1 \ X_1 \parallel_{X_2 \cup \dots \cup X_n} \ (\dots (P_{n-1} \ X_{n-1} \parallel_{X_n} \ P_n) \dots) \quad (3.10)$$

3.2.2.2 Trace semantics

The traces of $P \ X \parallel_Y \ Q$ are just those which combine a trace of P and a trace of Q so that all communications in $X \cap Y$ are shared.

$$\text{traces}(P \ X \parallel_Y \ Q) = \{s \in (X \cup Y)^* \mid s \upharpoonright X \in \text{traces}(P) \wedge s \upharpoonright Y \in \text{traces}(Q)\} \quad (3.11)$$

Where $s \upharpoonright X \in \text{traces}(P)$ means that the traces in s are restricted to the set X which are elements of the traces of P .

3.2.3 Interleaving

Interleaving allows its processes to run completely independent from each other. Any communicated event was either by process P or Q and no joined events are allowed. If both processes could communicate the same event, the choice of which one did so is nondeterministic and only one process could have done so at the given instance. More details regarding interleaving can be found in §2.3 of [2].

3.2.3.1 Laws

Step law

$$\begin{aligned}
 P \parallel Q &= ?x : (A \cup B) \rightarrow (P' \parallel Q) \cap (P \parallel Q') \\
 &\quad \langle x \in A \cap B \rangle \\
 &\quad (P' \parallel Q) \langle x \in A \rangle (P \parallel Q')
 \end{aligned} \tag{3.12}$$

Symmetry

$$P \parallel Q = Q \parallel P \tag{3.13}$$

Associativity

$$(P \parallel Q) \parallel R = P \parallel (Q \parallel R) \tag{3.14}$$

Distributivity

$$P \parallel (Q \cap R) = (P \parallel Q) \cap (P \parallel R) \tag{3.15}$$

Indexed notation

$$\parallel_{i=1}^n P_i = P_1 \parallel P_2 \parallel \dots \parallel P_{n-1} \parallel P_n \tag{3.16}$$

3.2.3.2 Trace semantics

The traces of $P \parallel Q$ are just the interleaved traces of P and Q . To calculate the interleaving traces, the following clauses are used.

$$\begin{aligned}
 \langle \rangle \parallel s &= \{s\} \\
 s \parallel \langle \rangle &= \{s\} \\
 \langle a \rangle \hat{\ } s \parallel \langle b \rangle \hat{\ } t &= \{ \langle a \rangle \hat{\ } u \mid u \in s \parallel \langle b \rangle \hat{\ } t \} \\
 &\quad \cup \{ \langle b \rangle \hat{\ } u \mid u \in \langle a \rangle \hat{\ } s \parallel t \}
 \end{aligned} \tag{3.17}$$

Given the recursive trace clauses of the interleaving operator in Equation 3.17, the traces of $P \parallel Q$ are given as:

$$\text{traces}(P \parallel Q) = \bigcup \{s \parallel t \mid s \in \text{traces}(P) \wedge t \in \text{traces}(Q)\} \tag{3.18}$$

3.2.4 Generalised parallel

The generalised parallel operator has become a commonly used parallel operator in model-checking tools although it has not been detailed in Hoare's initial CSP definition [11]. Roscoe [2] (§2.4) defined the generalised parallel operator as a short-hand notation for the alphabetised parallel operator. With alphabetised parallelism, $P \parallel_Y Q$, it is decided which events are synchronised and which are not by looking at each processes' alphabet, denoted by X and Y . The generalised parallel operator is given as $P \parallel_X Q$, where all events in X are synchronised and all events outside X can proceed independently. From this, the following hold:

$$P \parallel\parallel Q = P \parallel_{\{\}} Q \quad (3.19)$$

and, provided P and Q never communicate outside X and Y :

$$P \parallel_X \parallel_Y Q = P \parallel_{X \cap Y} Q \quad (3.20)$$

3.2.4.1 Laws

Step law

If $P = ?x : A \rightarrow P'$ and $Q = ?x : B \rightarrow Q'$ then:

$$\begin{aligned} P \parallel_X Q &= ?x : C \rightarrow (P' \parallel_X Q') \quad \langle x \in X \rangle \\ &(((P' \parallel_X Q) \sqcap (P \parallel_X Q')) \quad \langle x \in A \cap B \rangle \\ &((P' \parallel_X Q) \quad \langle x \in A \rangle \sqcap (P \parallel_X Q')) \end{aligned} \quad (3.21)$$

This can also be expressed as the external choice of four different processes. This is given in [17].

$$\begin{aligned} P \parallel_X Q &= (?x : X \cap A \cap B \rightarrow (P' \parallel_X Q')) \\ &\sqcap (?x : (A \cap B) \setminus X \rightarrow (P' \parallel_X Q) \sqcap (P \parallel_X Q')) \\ &\sqcap (?x : A \setminus (X \cup B) \rightarrow (P' \parallel_X Q)) \\ &\sqcap (?x : B \setminus (X \cup A) \rightarrow (P \parallel_X Q')) \end{aligned} \quad (3.22)$$

A Venn diagram can be constructed to indicate under which set conditions communication is allowed with generalised parallelism. Figure 3.1 shows the Venn diagram for generalised parallelism in CSP, where two processes P and Q communicate with each other over a non-empty synchronisation set X . In Figure 3.1, the excluded areas are indicated with a

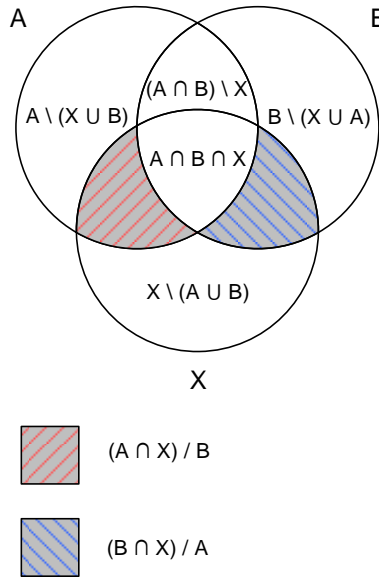


Figure 3.1: Venn diagram for sets of first actions in generalised parallelism according to [1].

greyed pattern. These are the areas where the step law of general parallelism is undefined, i.e. transitions are not allowed.

Symmetry

$$P \parallel_X Q = Q \parallel_X P \quad (3.23)$$

Associativity

$$(P \parallel_X Q) \parallel_X R = P \parallel_X (Q \parallel_X R) \quad (3.24)$$

Distributivity

$$P \parallel_X (Q \sqcap R) = (P \parallel_X Q) \sqcap (P \parallel_X R) \quad (3.25)$$

Indexed notation

$$\parallel_{X \ i=1}^n P_i = P_1 \parallel_X P_2 \parallel_X \dots \parallel_X P_{n-1} \parallel_X P_n \quad (3.26)$$

3.2.4.2 Trace semantics

The traces of $P \parallel_X Q$ are the combinations traces of P and Q where actions in X are shared and all other occur independently. To calculate the traces of $s \parallel_X t$ for all $s, t \in \Sigma^*$, where x denotes a member of X and y a member of $\Sigma \setminus X$, the following clauses from [2] (p. 69) are used:

$$\begin{aligned}
 s \parallel_X t &= t \parallel_X s \\
 \langle \rangle \parallel_X \langle \rangle &= \{\langle \rangle\} \\
 \langle \rangle \parallel_X \{x\} &= \langle \rangle \\
 \langle \rangle \parallel_X \{y\} &= \{\langle y \rangle\} \\
 \langle x \rangle \wedge s \parallel_X \langle y \rangle \wedge t &= \{\langle y \rangle \wedge u \mid u \in \langle x \rangle \wedge s \parallel_X t\} \\
 \langle x \rangle \wedge s \parallel_X \langle x \rangle \wedge t &= \{\langle x \rangle \wedge u \mid u \in s \parallel_X t\} \\
 \langle x \rangle \wedge s \parallel_X \langle x' \rangle \wedge t &= \{\} \quad \text{if } x \neq x' \\
 \langle y \rangle \wedge s \parallel_X \langle y' \rangle \wedge t &= \{\langle y \rangle \wedge u \mid u \in s \parallel_X \langle y' \rangle \wedge t\} \\
 &\quad \cup \{\langle y' \rangle \wedge u \mid u \in \langle y \rangle \wedge s \parallel_X t\}
 \end{aligned} \tag{3.27}$$

Given the recursive trace clauses of the generalised parallel operator in Equation 3.27, the traces of $P \parallel_X Q$ are given as:

$$\text{traces}(P \parallel_X Q) = \bigcup \{s \parallel_X t \mid s \in \text{traces}(P) \wedge t \in \text{traces}(Q)\} \tag{3.28}$$

3.2.5 Optional parallel

Optional parallelism [17] allows processes to synchronise on shared events just as generalised parallelism, but also to proceed independently when an event in the synchronisation set is to be communicated and not available in one of the processes communicating under optional parallelism. The behaviour of optional parallelism is obtained by lifting the restriction in the

step laws of generalised parallelism where all processes need to jointly synchronise in common communication events. These are the shaded areas of the Venn diagram of Figure 3.1.

The operational semantics of $OptPar$ has been given in [17] in the style of the step laws of Roscoe [2]. The approach followed in this dissertation is to model the behaviour of optional parallelism given in [17] using existing CSP operators, providing a *translational* semantics. The $\overset{\sim}{\parallel}_X$ operator will be used for this translation, referred to as $OptPar^T$, which is detailed in Chapter 4. This allows the semantics of the existing operators to be inherited which enables the new operator to be used as *syntactic sugar* in models where concurrent processes cannot always jointly engage in synchronisation. This approach allows optional parallelism to be used in CSP models without defining it from first principles.

The following relation exists between optional parallelism and interleaving [17]:

$$P \overset{\sim}{\parallel}_{\{\}} Q = P \parallel Q \quad (3.29)$$

3.2.5.1 Laws

Step law

If $P = ?x : A \rightarrow P'$ and $Q = ?x : B \rightarrow Q'$ then:

$$\begin{aligned}
 P \overset{\sim}{\parallel}_X Q &= ?x : C \rightarrow (P' \overset{\sim}{\parallel}_X Q') \langle x \in X \rangle \\
 &(((P' \overset{\sim}{\parallel}_X Q) \sqcap (P \overset{\sim}{\parallel}_X Q')) \langle x \in A \cap B \rangle \\
 &((P' \overset{\sim}{\parallel}_X Q) \langle x \in A \rangle \\
 &((P \overset{\sim}{\parallel}_X Q') \langle x \in B \rangle \\
 &((P' \overset{\sim}{\parallel}_X Q) \langle X \cap A \rangle (P \overset{\sim}{\parallel}_X Q')))) \quad (3.30)
 \end{aligned}$$

This can also be expressed as the external choice of six different processes. This is given in [17].

$$\begin{aligned}
 P \uparrow_X Q &= (?x : X \cap A \cap B \rightarrow (P' \uparrow_X Q')) \\
 &\sqcap (?x : (A \cap B) \setminus X \rightarrow (P' \uparrow_X Q) \sqcap (P \uparrow_X Q')) \\
 &\sqcap (?x : A \setminus (X \cup B) \rightarrow (P' \uparrow_X Q)) \\
 &\sqcap (?x : B \setminus (X \cup A) \rightarrow (P \uparrow_X Q')) \\
 &\sqcap (?x : (X \cap A) \setminus B \rightarrow (P' \uparrow_X Q)) \\
 &\sqcap (?x : (X \cap B) \setminus A \rightarrow (P \uparrow_X Q'))
 \end{aligned} \tag{3.31}$$

This can be simplified to:

$$\begin{aligned}
 P \uparrow_X Q &= (?x : X \cap A \cap B \rightarrow (P' \uparrow_X Q')) \\
 &\sqcap (?x : (A \cap B) \setminus X \rightarrow (P' \uparrow_X Q) \sqcap (P \uparrow_X Q')) \\
 &\sqcap (?x : A \setminus B \rightarrow (P' \uparrow_X Q)) \\
 &\sqcap (?x : B \setminus A \rightarrow (P \uparrow_X Q'))
 \end{aligned} \tag{3.32}$$

From [17], *OptPar* is defined to behave the same as generalised parallelism of CSP, with the addition of step law choices where generalised parallelism deadlocks. The generalised parallel operator's step law is defined in Equation 3.22 and it follows that steps where $x \in ((X \cap A) \setminus B) \cup ((X \cap B) \setminus A)$ does not allow any progress of the two processes in parallel. Optional parallelism allows progress when the environment presents these events. This allows processes under optional parallelism to *always* engage in events presented by the environments, essentially eliminating deadlock from the system.

Symmetry

$$P \uparrow_X Q = Q \uparrow_X P \tag{3.33}$$

Associativity

$$(P \uparrow_X Q) \uparrow_X R = P \uparrow_X (Q \uparrow_X R) \tag{3.34}$$

Distributivity

$$P \uparrow_X (Q \sqcap R) = (P \uparrow_X Q) \sqcap (P \uparrow_X R) \quad (3.35)$$

Indexed notation

$$\prod_{X, i=1}^n P_i = P_1 \uparrow_X P_2 \uparrow_X \dots \uparrow_X P_{n-1} \uparrow_X P_n \quad (3.36)$$

3.2.5.2 Trace semantics

The traces of $P \uparrow_X Q$ are the combinations traces of P and Q where actions in X are shared or occur independently. To calculate the traces of $s \uparrow_X t$ for all $s, t \in \Sigma^*$, where $\langle \rangle$ denotes the empty trace and \checkmark the successful termination event that is per definition not in the alphabet Σ , the following clauses from [17] are used:

$$\begin{aligned} s \uparrow_X t &= t \uparrow_X s \\ \langle \rangle \uparrow_X t &= \{t\} \\ \langle a \rangle \uparrow_X s \parallel \langle b \rangle \uparrow_X t &= \begin{cases} \{\langle a \rangle \hat{\ } u \mid u \in s \uparrow_X t\} & \text{if } a = b \wedge a \in X \\ \{\langle a \rangle \hat{\ } u \mid u \in s \uparrow_X \langle b \rangle \hat{\ } t\} \cup \{\langle b \rangle \hat{\ } u \mid u \in \langle a \rangle \hat{\ } s \uparrow_X t\} & \text{if } a \neq b \vee a \notin X \end{cases} \\ s \hat{\ } \langle \checkmark \rangle \uparrow_X t &= \emptyset \\ s \uparrow_X t \hat{\ } \langle \checkmark \rangle &= \emptyset \\ s \hat{\ } \langle \checkmark \rangle \uparrow_X t \hat{\ } \langle \checkmark \rangle &= \{u \hat{\ } \langle \checkmark \rangle \mid u \in s \uparrow_X t\} \end{aligned} \quad (3.37)$$

Given the trace clauses of *OptPar* in Equation 3.37, the traces of $P \uparrow_X Q$ are given as:

$$traces(P \uparrow_X Q) = \bigcup \{s \uparrow_X t \mid s \in traces(P) \wedge t \in traces(Q)\} \quad (3.38)$$

3.3 CHAPTER SUMMARY

This chapter serves the purpose to give the reader a reference of the CSP syntax and the operators used in this dissertation. The details are mainly extracted from [2] and [17]. Many of the chapters that follow will reference the laws and equations given in this chapter.

CHAPTER 4

DEFINITION OF A NEW OPTIONAL PARALLEL OPERATOR

4.1 PROBLEM IDENTIFICATION

Two common problems found in concurrent systems are deadlock and livelock. Deadlock occurs when none of the processes of a concurrent system can make any progress, which typically happens when there is a cycle of events waiting for each other to complete before making progress. This is common in a shared resource environment. An example of such an occurrence in the real world is where 4 cars with polite motorists simultaneously arrive at a 4-way stop and each of the motorists are waiting for the other motorists to cross the intersection first. The cars are deadlocked because no motorist will cross the intersection before the other. The competition for resources, in this case the intersection, is one of the main causes of deadlock [2]. Deadlock occurs when all of the following conditions, called the Coffman conditions [108], arise in a system.

- Mutual exclusion - A resource may be used by only one process at a time.
- Hold and Wait - Some of the required resources of a process may be allocated, while the process waits for the other resources to become available.
- No pre-emption - A resource cannot be removed from a process holding it, the process must voluntarily release it.
- Circular wait - Processes wait on each other in such a way that a cycle of wait conditions

exists. Process P waits for Q, Q waits for R and R waits for P.

Livelock occurs when a process performs infinite internal actions without communicating with its environment, which is also referred to as divergence. From a software programming perspective, this is referred to as infinite loops. A concurrent system of processes has a livelock condition when the processes infinitely perform internal actions without any external interaction. From an observer's perspective, the system seems deadlocked, which is clearly not the case. The most common real world example is where two persons approach each other on a walkway. When they want to pass each other, they both sway to the same side together and then to the other side together. The two persons are in a livelock condition, they both perform actions (swaying left and right), but none of them progress past each other.

Livelock properties cannot be checked with the *traces* model \mathcal{T} of CSP and more complex models like the *stable failures* model \mathcal{F} or the *failures/divergences* model \mathcal{N} are needed. Instead, safety properties should be checked using the *traces* model. Safety can be described as "nothing bad will ever happen", in contrast to liveness properties which indicate that "eventually, something good will happen". Safety properties are defined as a sequence of traces and this sequence can then be used in trace refinement checks to see if the property can occur if not. This is confirmed by Roscoe [2] where it is stated that trace specifications cannot force a system to do anything and are referred to as *safety* or *partial correctness* conditions while *liveness* or *total correctness* conditions are capable of forcing a process to do something. With this in mind, livelock will not be addressed by this dissertation because of the use of the *traces* model. Safety properties are checked with the trace refinement tests used later in this dissertation. It is still useful to do deadlock analysis because it can certainly occur even if only the *traces* model is used for system definitions.

The following examples used to identify the problem are based on the high level observations mentioned in [17], where optional parallelism was first introduced.

Consider a small WSN with nodes P , Q and R . This can be modelled in CSP by placing the nodes in parallel as in Equation 4.1.

$$\begin{aligned}
 WSN_{PQR} &:= P \parallel_X Q \parallel_X R \\
 X &:= \alpha P \cap \alpha Q \cap \alpha R
 \end{aligned}
 \tag{4.1}$$

Say a WSN node measures humidity and temperature, represented by $senseH$ and $senseT$. A node sends its particular measured parameter after each measurement, represented by $sendH$ and $sendT$ ¹. The definitions of the nodes can be seen in Equation 4.2.

$$\begin{aligned}
 P &:= senseH_p \rightarrow sendH \rightarrow senseT_p \rightarrow sendT \rightarrow P \\
 Q &:= senseH_q \rightarrow sendH \rightarrow senseT_q \rightarrow sendT \rightarrow Q \\
 R &:= senseH_r \rightarrow sendH \rightarrow senseT_r \rightarrow sendT \rightarrow R \\
 X &:= \alpha P \cap \alpha Q \cap \alpha R = \{sendH, sendT\}
 \end{aligned} \tag{4.2}$$

The WSN_{PQR} system is deadlock free, with each process jointly synchronising on the $sendH$ and $sendT$ events. If node P' represents a WSN node where its humidity sensor can fail, the behaviour can be modelled as given in Equation 4.3. The WSN system with this definition of process P' becomes $WSN_{P'QR}$.

$$P' := (senseH_p \rightarrow sendH \rightarrow senseT_p \rightarrow sendT \rightarrow P') \sqcap (senseT_p \rightarrow sendT \rightarrow P') \tag{4.3}$$

$WSN_{P'QR}$ then has a deadlock condition if node P' has a humidity sensor failure and performs $senseT_p$ first. Processes Q and R will block to synchronise on $sendH$, while node P' blocks to perform $sendT$. The blocking is due to the requirement of joint synchronisation on the same event under parallelism in CSP.

Another scenario that could occur is that node P has the ability to sleep and wake at a specific time to perform measurements and transmit its parameters. The definition for node P'' is shown in Equation 4.4.

$$P'' := (senseH_p \rightarrow sendH \rightarrow senseT_p \rightarrow sendT \rightarrow P'') \sqcap (sleep_p \rightarrow P'') \tag{4.4}$$

This introduces non-determinism as it cannot be guaranteed that node P'' ever wakes up to synchronise with nodes Q and R . In this case, nodes Q and R will block indefinitely.

Expanding the example of $WSN_{P'QR}$ where the nodes need to send their measurements to a sink node S , the WSN is modelled as shown in Equation 4.5.

$$\begin{aligned}
 WSN_{P'QRS} &:= S \parallel_X (P' \parallel_X Q \parallel_X R) \\
 X &:= \alpha P' \cap \alpha Q \cap \alpha R \cap \alpha S
 \end{aligned} \tag{4.5}$$

¹Note that this example, as like the rest of this dissertation, abstracts from transmitting data, i.e. sensed values. The examples are trivial and at a systems-level of WSN nodes to simplify the concepts of the optional parallel operator and its translation to classical CSP operators.

The sink node process definition is given in Equation 4.6.

$$S := (\text{send}H \rightarrow S) \square (\text{send}T \rightarrow S) \quad (4.6)$$

Then, in the case where node P' has a humidity sensor failure, the system will deadlock, even if P' is ready to send its measured temperature to the sink node S and both Q and R are ready to send their humidity measurements to the sink S . The same happens when node P' is replaced by the definition of P'' to form $WSN_{P''QRS}$. This occurs because the nodes are required to jointly synchronise on the same events in X .

A possible solution to the problem is to make nodes P , Q and R independent by modelling the WSN with the interleaving operator. The system definition of Equation 4.5 is modified by exchanging the general parallel operator with an interleaving operator as shown in Equation 4.7

$$WSN_{P'QRS}^{Inter} := S \parallel_X (P' \parallel Q \parallel R)$$

$$X := \alpha S \cap (\alpha P' \cup \alpha Q \cup \alpha R) \quad (4.7)$$

In the WSN system of Equation 4.7, there will always be a node available to synchronise with the sink S . This has the problem that at most one node is able to synchronise with the sink S at any time, which is not the desired result. There is a need to model systems where a subset of processes are able to jointly engage on a synchronisation event.

4.2 OPTIONAL PARALLELISM

The restriction of the parallel operator, which states that all processes under parallelism should jointly engage on the same synchronisation event to proceed, limits the expressiveness of WSN system definitions like Equation 4.5. There is currently no simple way of modelling a system where only a subset of the node processes can synchronise with the sink. The requirement is a single CSP operator which allows its processes to communicate events from its synchronisation set even if there are no other processes to synchronise with.

Consider the WSN system example of Equation 4.5. If the node processes could be modelled in such a way that the sink S can synchronise with any subset of the nodes which are ready, the problem is solved. This allows the behaviour where a broadcast communication event

can originate from the sink to all of its connected nodes and only the nodes which are ready, will receive the message.

With the synchronisation restriction lifted, all or any subset of processes under optional parallelism are allowed to jointly synchronise on a common synchronisation event. Initial work on optional parallelism ($OptPar$) has been presented in [17], and the solution in this dissertation defines optional parallelism by making use of classical CSP operators ($OptPar^T$). This approach is followed because a model-checker with an optional parallel implementation does not exist. The new definition of optional parallelism, $OptPar^T$, will be given the symbol $\tilde{\parallel}$.

4.2.1 Defining optional parallelism using classical CSP operators

When using classical CSP operators to model the behaviour of optional parallelism in the traces domain, the existing laws of the operators are inherited and should be adhered to. A combination of the parallel and interleaving operators will be used, as optional parallelism describes a hybrid functionality of both these operators. To allow a subset of processes to synchronise with the sink has two effects:

1. The processes should be independent from each other, hence the use of the interleaving operator.
2. The processes should be allowed to synchronise jointly, even if they are not ready. As mentioned earlier, the joint synchronisation is a restriction of the CSP parallel operator.

By using the example of a broadcasting process and only a subset of processes receiving the broadcast, a channel modelling artefact is introduced, which always receives the broadcast, but decides to pass the message to the receiving process only if the process is ready to receive. This will have the effect that the broadcaster can perform a synchronisation event, and all *channels* jointly engage in the synchronisation, but the receiving processes are not required to, as they are disconnected from each other. There will always be a joint synchronisation and the processes are independent of each other. This covers the two effects of modelling optional parallelism with classical CSP operators.

4.2.1.1 Channel artefact

The approach of $OptPar^T$ is based on lossy channels and the Alternating Bit Protocol (ABP) [2, 42]. The ABP is used to overcome the difficulties of lossy channels where messages are re-transmitted until they are acknowledged. The process structure definition of the ABP given in [2] is similar to the structure of the channel artefact. $OptPar^T$ involves the addition of an artificial channel modelling artefact, prone to errors, to each common synchronisation event between connected processes. The channel is modelled to guarantee synchronisation under general parallelism with its attached processes, and is further able to allow or drop inbound or outbound communication events. The approach followed here differs from the ABP in the sense that duplication of messages are not defined. Time-outs and acknowledge messages are also not considered in the channel definitions as this is a time independent solution, as are most CSP definitions.

When the channel artefact is modelled with the WSN node process as a unit, the node process will have no knowledge if a connected receiver process has received the communication event or not. It is disconnected from its neighbouring node processes through the channels between them and is therefore not dependent on the other node processes' joint synchronisation. The synchronisation is orchestrated only by the channel artefacts. The CSP representation is given in Equation 4.8.

$$\begin{aligned}
 C &:= (internal \rightarrow ((external \rightarrow C) \square (drop \rightarrow C))) \\
 &\square (external \rightarrow ((internal \rightarrow C) \square (drop \rightarrow C))) \setminus \{drop\}
 \end{aligned} \tag{4.8}$$

The channel artefact definition of Equation 4.8 represents forward and backward communication. One for when its parent process communicates with the environment, and one where the environment communicates with the parent process. It is defined to always be available to perform an *internal* or an *external* event. The *internal* event represents the exclusive event between a process and its channel. The common synchronisation events in process definitions will be renamed to this *internal* event name. The *external* event represents the common synchronisation event between the channel processes of the network. The *drop* event is used to indicate that a synchronisation has been dropped by the channel due to either the process or the environment not being ready to synchronise. The *drop* event is hidden from external observation by hiding it. The choice to drop communication or forwarding it is non-

deterministic from the view of the sending node process. The external choice operator \square is chosen because the channel is obliged to offer events *external* or *drop* after the *internal* event, depending on the state of the receiving node process. The use of the internal choice operator \sqcap requires that both *external* and *drop* need to be offered and a choice is made internally by the channel artefact on which event will be next. In this approach, a communication event should not happen if the receiving process cannot receive it, which is the main argument behind the use of external choice to model the channel artefact. The channel artefact has a simple definition, unlike in [2] where divergence is introduced with infinite duplication. C is not considered to be a buffer because it does not satisfy all the requirements of a buffer, i.e. it will not always output what has been input. It is also structurally simpler than a buffer. Note that the *drop* event will be explicitly omitted in future channel definitions of this dissertation.

4.2.1.2 Inserting channel artefacts into system definitions

The channel artefacts need to be added to the system definition to allow optional parallel behaviour between the processes. Each process gets a channel, attached in parallel, for every event in its synchronisation set. Consider the WSN network graph in Figure 4.1. Processes P , Q and R are all connected with a hyper-edge, which represents a common synchronisation event $comm$, between them. When one of the processes wants to synchronise on $comm$, all of the other processes have to join on $comm$. For this example, the internal structure of the processes are arbitrary, and it is only known that each process performs a $comm$ event when it is ready to do so.

When a channel artefact is added to each process, the graph transforms into the one shown in Figure 4.2. The channels CHP , CHQ and CHR are inserted to disconnect the processes from each other. The $comm$ events of the processes P , Q and R are renamed to $comm_p$, $comm_q$ and $comm_r$ respectively and the original synchronisation event $comm$ is still observable from the environment through the added channels. The *internal* events of CHP , CHQ and CHR are renamed to $comm_p$, $comm_q$ and $comm_r$ respectively and their *external* events renamed to $comm$. This is shown by renaming the events of Equation 4.8 for each channel to its

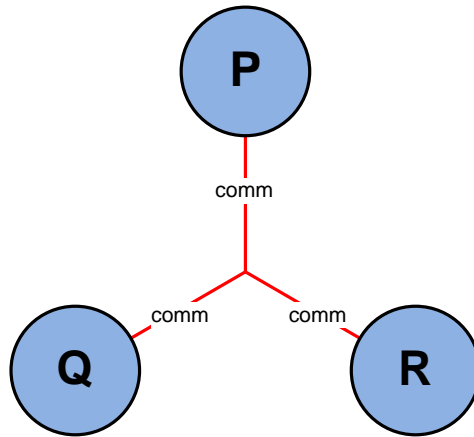


Figure 4.1: Basic WSN network graph with hyper-edge.

corresponding synchronisation events as shown for CHP below in Equation 4.9

$$\begin{aligned}
 CHP &:= (comm_p \rightarrow ((comm \rightarrow CHP) \square CHP)) \\
 &\square (comm \rightarrow ((comm_p \rightarrow CHP) \square CHP))
 \end{aligned}
 \tag{4.9}$$

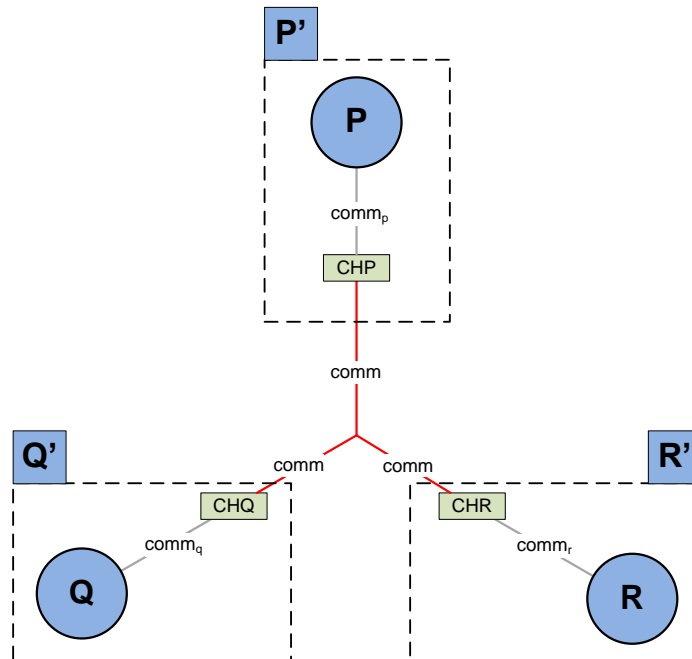


Figure 4.2: Basic network graph with hyper-edge and added channel artefacts.

CHP is defined to always be available for *internal comm_p* events as well as *external comm* events. If P wants to perform its synchronisation event, now renamed to *comm_p*, it synchronises with its channel CHP , which translates the *comm_p* event to the global synchronisation *comm* event. The channel will not allow the *comm* event if all the other channels are not ready, effectively eliminating possible collisions. If another process wants to synchronise with P on the global *comm* event, CHP does the initial synchronisation on *comm* to ensure joint synchronisation, and then, depending on the internal state of P , tries to synchronise with P on *comm_p*, or drops the communication. The channel will drop the communication if *comm_p* is not offered by P , which emphasises the argument for using external choice of the channel artefact definition of Equation 4.8. This has the effect that P' opts out of communication, while still obeying the laws of parallelism in CSP, i.e. joint synchronisation. Channels CHQ and CHR have the same behaviour as CHP .

The processes are modelled with their channels as in Equation 4.10.

$$\begin{aligned}
 P' &:= P \parallel_{\{comm_p\}} CHP \\
 Q' &:= Q \parallel_{\{comm_q\}} CHQ \\
 R' &:= R \parallel_{\{comm_r\}} CHR
 \end{aligned} \tag{4.10}$$

The WSN of P , Q and R of Figure 4.2 is modelled with the primes of each process as shown in Equation 4.11.

$$WSN_{PQR}^{optional} := P' \parallel_{\{comm\}} Q' \parallel_{\{comm\}} R' \tag{4.11}$$

In the case of the addition of a sink node S to the WSN definition, the desired behaviour of optional parallelism is achieved because S will be able to synchronise with all other processes or a subset thereof. The CSP definition, which resolves the problems identified in Equations 4.5 and 4.7, is shown in Equation 4.12

$$WSN_{P'Q'R'S}^{optional} := S \parallel_{\{comm\}} (P' \parallel_{\{comm\}} Q' \parallel_{\{comm\}} R') \tag{4.12}$$

The addition of channel artefacts to each process enables the initial system of Equation 4.1 to be modelled with optional parallelism for the cases where WSN nodes can have failures

as in Equation 4.3 and where the nodes are modelled in such a way where divergence is present as in Equation 4.4. The WSN example definitions of Equations 4.11 and 4.12 can be modelled by using the new optional parallel operator, which encapsulates the addition of a channel artefact for each element in the synchronisation set between processes. This allows an easy transition from system definitions using the general parallel operator of CSP where the parallel operators are replaced with optional parallel operators where optional parallelism is required. This is shown in Equation 4.13 for Equation 4.12

$$S \parallel_X (P' \parallel_X Q' \parallel_X R') = S \parallel_X (P \tilde{\parallel}_X Q \tilde{\parallel}_X R)$$

$$X := \alpha P \cap \alpha Q \cap \alpha R \cap \alpha S \quad (4.13)$$

Note that the alphabet X stays the same for both ends of the equation and the external events observed will stay the same between the two definitions.

4.2.2 Preparing system definitions for optional parallelism analysis

OptPar is not currently defined in any model-checking tool. Systems with *OptPar* requirements can be converted to systems using classical CSP operators with *OptPar^T*, which can be model-checked with existing model-checkers. One of the main design requirements is deadlock freeness, which is one of the core problems which optional parallelism attempts to solve, hence the testing thereof later in the evaluation of *OptPar^T*.

In order to build the foundation for CSP deadlock analysis and to build on existing CSP mechanisms like binary relationships, the following properties from [2] should be assumed of the WSN.

- None of the component processes can terminate, i.e. they execute perpetually.
- The network is statically defined, i.e. $\parallel_{i=1}^n (P_i, A_i)$ where the communication of P_i are entirely within A_i .
- Network is triple-disjoint, i.e. there is no event that requires more than two component processes. More formally, if P_i , P_j and P_k are three distinct processes in the network,

$$A_i \cap A_j \cap A_k = \{\}$$

- The network is built using parallel composition from their component processes and contain no renaming or hiding.
- The network is busy, meaning each component process is deadlock free.

From the properties above it is given that the WSN needs to be triple disjoint, which means that for the edges $E = \{e_1..e_n\}$, $|e_i| = 2$ and $1 \leq i \leq n$. This means that each edge must have a depth of 2, which implicates binary relations.

From Figure 4.2 it can be seen that nodes P' , Q' and R' are fully connected with each other and forming a hyper-edge of three nodes. The hypergraph equation of the WSN in is given in Equation 4.14.

$$\begin{aligned} WSN &:= (N, E) \\ N &:= \{P', Q', R'\} \\ E &:= \{\{P', Q', R'\}\} \end{aligned} \quad (4.14)$$

In [109] it is shown that an n-ary relation can be converted to a set of binary relations without losing information. Roscoe [2] confirms this with the following relation:

$$\left\|_{i=1}^3 P_i = (P_1 \parallel P_2 \parallel P_3) = ((P_1 \parallel P_2) \parallel P_3) = (P_1 \parallel (P_2 \parallel P_3)) \quad (4.15)$$

Based on these reasons, the hypergraph of Equation 4.14 can be converted to a normal graph with binary relations as shown in Figure 4.3.

The new graph definition becomes:

$$\begin{aligned} WSN &:= (N, E) \\ N &:= \{P', Q', R'\} \\ E &:= \{\{P', Q'\}, \{P', R'\}, \{Q', R'\}\} \end{aligned} \quad (4.16)$$

The normal graph of Figure 4.3 can further be modified to simplify its analysis from a modelling perspective. Although the normal graph solution of Figure 4.3 and Equation 4.16 satisfies the required triple disjointness of [2] for deadlock analysis, it can further be

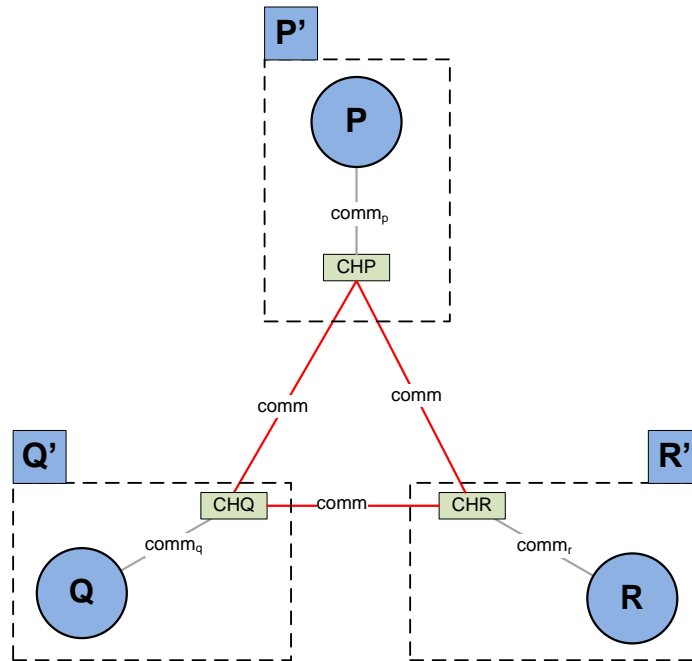


Figure 4.3: Normal graph representation of the hypergraph of Figure 4.2.

modified to allow for it to be easily applicable to WSN graph topologies with normal graph definitions, i.e. binary connections between WSN nodes.

The conversion of the WSN graph of Figure 4.3 to a graph containing binary relationships between the nodes entails the graphical duplication of the channel artefacts of a node to have one for each connected node sharing the same synchronisation event. After this duplication, the channel artefacts sharing the same binary connection between nodes are grouped together to form a single channel artefact between two connected nodes. This is shown in Figure 4.4. Although the graph of Figure 4.4 is graphically different from the graph of Figure 4.3, nothing has changed in its CSP system definition except for the composition of the concurrent processes, which will become apparent in the generalised $OptPar^T$ definition of Equation 4.21.

The alphabetised parallel operator is used in the CSP system definitions because it keeps track of each process' alphabet when computing the synchronisation sets between processes. This is also confirmed by [2] in the needed assumptions for deadlock analysis. Alphabetised parallel allows for more practical compositions at the expense of more streamlined definitions like the

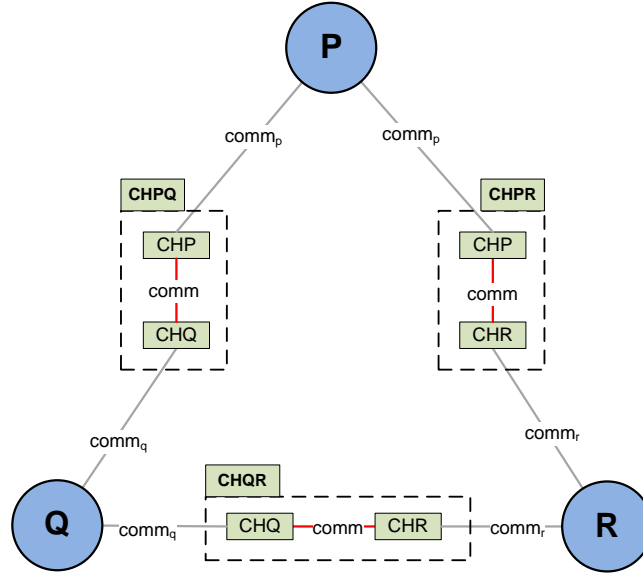


Figure 4.4: Binary relationship conversion of Figure 4.3.

use of the generalised parallel operator. The alphabets of the processes under alphabetised parallelism need to be kept track of, while generalised parallelism is only concerned with one set on intersections of all processes' alphabets, hence the streamlined notation. Equation 3.10 gives the alphabetised parallel operator in an indexed notation for n-way parallel composition. Using the index form of alphabetised parallelism of Equation 3.10, the WSN of Figure 4.4 is defined as in Equation 4.17.

$$\begin{aligned}
 WSN &:= \parallel_{i=1}^6 (P_i, \alpha P_i), \text{ where} \\
 (P_1, \alpha P_1) &= (P, \alpha P) \\
 (P_2, \alpha P_2) &= (Q, \alpha Q) \\
 (P_3, \alpha P_3) &= (R, \alpha R) \\
 (P_4, \alpha P_4) &= (CHPQ, \alpha CHPQ) \\
 (P_5, \alpha P_5) &= (CHPR, \alpha CHPR) \\
 (P_6, \alpha P_6) &= (CHQR, \alpha CHQR)
 \end{aligned} \tag{4.17}$$

where αP_i represents the alphabet of possible communications of process P_i .

When this is expanded as per the definition of Equation 3.10, some simplifications can be made to the model. When the node processes' parallel interaction is viewed separately from the channel processes, their synchronisation sets are all empty. This is shown in Equation 4.18 and Equation 4.19.

$$\begin{aligned}
 Nodes &:= \parallel_{i=1}^3 (P_i, \alpha P_i) \\
 &= P_1 \ \alpha P_1 \parallel_{\alpha P_2 \cup \alpha P_3} (P_2 \ \alpha P_2 \parallel_{\alpha P_3} P_3)
 \end{aligned} \tag{4.18}$$

Substituting the process names and alphabets has the effect of interleaving because none of the processes share communication events as there are channel artefacts defined to exist between the processes. This is clear from Figure 4.4. Using the trace law of alphabetised and the general parallel operators' properties defined in [2], the definition in Equation 4.18 is reduced to:

$$\begin{aligned}
 Nodes &:= P \ \alpha P \parallel_{\alpha Q \cup \alpha R} (Q \ \alpha Q \parallel_{\alpha R} R) \\
 &= P \parallel_{\alpha P \cap (\alpha Q \cup \alpha R)} \parallel_{\alpha Q \cap \alpha R} (Q \parallel R) \\
 &= P \parallel_{\{\}} \parallel_{\{\}} (Q \parallel R) \\
 &= P \parallel \parallel (Q \parallel \parallel R) \\
 &= P \parallel \parallel Q \parallel \parallel R
 \end{aligned} \tag{4.19}$$

The channel artefacts, however, do not share the same disconnectedness as the node processes, although it seems as if they are disconnected from each other in Figure 4.4. This is because of the shared channel artefacts of the processes in the binary relationships. Channel artefact CHP is contained in both $CHPQ$ and $CHPR$ and thus creates a dependency between $CHPQ$ and $CHPR$. This has no restriction on the expressiveness of optional parallelism using classical CSP operators, and the binary channel artefacts $CHPQ$, $CHPR$ and $CHQR$ are modelled with the alphabetised parallel operator as in Equation 4.17. This shows that the node processes are fully disconnected from each other, with only the channel artefacts being allowed to orchestrate the synchronisations between node processes. The definition of

the WSN can be given as:

$$\begin{aligned}
 WSN &:= Nodes \parallel_{\alpha N} Channels, \text{ where} \\
 Nodes &:= \parallel_{i=1}^n N_i \\
 Channels &:= \parallel_{j=1}^c (C_j, \alpha C_j)
 \end{aligned} \tag{4.20}$$

4.2.3 Generalisation of optional parallelism with classical CSP operators

The definition of the WSN, and thus the translation of the behaviour of optional parallelism into existing CSP operators is shown in Equation 4.21.

$$\begin{aligned}
 \tilde{\parallel}_{i=1}^n (N_i, \alpha N_i) &:= \left(\parallel_{i=1}^n N_i \right) \parallel_X \left(\parallel_{j=1}^c (C_j, \alpha C_j) \right) \\
 X &:= \left(\bigcup_{i=1}^n \alpha N_i \right) \cap \left(\bigcup_{j=1}^c \alpha C_j \right)
 \end{aligned} \tag{4.21}$$

With N_i representing all the processing nodes wishing to synchronise, and C_j the channel definitions for each synchronisation element between all of the processes. The channel process definitions are selected based on the required type of synchronisation indicated with a directional notation of the synchronisation events of the processes. This is explained in §4.3 and a detailed example is given in §6.1.1.1.1. In short, the process definitions on the left hand side of Equation 4.21 undergoes renaming to form the process definitions on the right hand side. The channel definitions are added as a result of the $OptPar^T$ operator $\tilde{\parallel}$ and the type of directional synchronisation indicated on the synchronisation events of the processes on the left hand side of Equation 4.21. From a graph perspective, $\{N_i \mid 1 \leq i \leq n\}$ indicates the vertices V and $\{C_j \mid 1 \leq j \leq c\}$ the edges E . This can be directly related to the example of Equation 4.16.

The definition of optional parallelism in terms of classical CSP operators in Equation 4.21 indicates that all the node processes are independent from each other, whereas relationship dependencies exist between the channel artefacts. The use of the central parallel operator, here given as generalised parallel for expressiveness, between the group of node processes and the group of channel artefacts, ensures that the node processes are well behaved and their combination are divergence free [2]. The channel artefacts act as a monitor and restricts

the behaviour of the network, effectively eliminating livelock. The synchronisation set X is defined to be the intersection between the union of alphabets of all the node processes and the union of alphabets of the channel artefacts. This effectively breaks the common synchronisation event, which would have been present in generalised parallel as in Equation 4.1, up into individual synchronisation events, one for each node process. The interest lies in the shared events which can be given as $\cup\{A_i \cap A_j | i \neq j\}$ where A represents all processing elements in the network. This is referred to as the vocabulary of the network. Furthermore, all node processes connected with a channel artefact can synchronise, but if synchronisation is not possible, the node processes can run independently from each other.

Equation 4.21 has the following mapping to a WSN:

- N_i represents only the WSN nodes.
- The channels are automatically added based on the intersections of the alphabets, αN_i , of the nodes.
- There exists a channel C_j for each common synchronisation event between each of the node processes N_i .
- The definitions of the processes can vary, but the channel definitions are fixed as per Equation 4.8.
- The alphabet αC_j of each channel C_j is made up from the synchronisation events of the two node processes N_i and N_k , $i \neq k$ connected to each side of the channel C_j .

The topology structure of the WSN is contained in two parts of Equation 4.21. The first part is the number of nodes present, defined by N_i , and the second part is defined in the combined channel artefacts $\{C_j | 1 \leq j \leq c\}$, which indicates which processes can communicate with each other. The alphabetised parallel structure of the channel artefacts C_j allows only the channel artefacts with non-empty intersections of their alphabets to synchronise with each other, which implicitly encapsulates the topology structure of the network.

4.3 DIRECTIONAL SYNCHRONISATION FOR OPTIONAL PARALLELISM USING CHANNEL ARTEFACTS

The previous section focussed on fully connected networks where all processes share the same synchronisation event. All nodes communicating under optional parallelism were allowed to synchronise on a communication event, no matter from which process it is from. Broadcasting is an example where this behaviour is not desirable. A broadcaster process synchronises with its neighbour processes during a broadcast communication, but the neighbouring processes are not allowed to synchronise with each other. In other words, information is passed from the broadcaster to its neighbours, but not between the neighbours.

This is better described by example. Consider the cluster network topology of Figure 4.5. It has a cluster head P and a set of neighbour nodes Q , R and S to which the cluster head is broadcasting some data. The neighbour nodes are not connected to each other, so if a broadcast message is transmitted from the cluster head, all neighbour nodes receive the message, but when a neighbour node transmits a message, only the central node will receive it individually. This has the implication that the dataflow is directional. With the CSP analysis being data independent, the only way of indicating direction is by enabling a process definition to initiate an event on its own, or by responding to an event from another process. This is done by adding a channel artefact for each *receiving* event to each process with the particular event. The definition of the channel artefact restricts its parent process from initiating a communication by only allowing to synchronise on the parent process' communication event after one has been received from the cluster head.

The CSP system definition is given in Equation 4.22. Here, the cluster head senses some environment variable and communicates it to its neighbours. Each of the neighbour processes performs some work or calculations on the received measurement from the cluster head. Each neighbour node is also allowed to go into a sleep state to save energy. When the cluster head broadcasts its measurement to the neighbour nodes, it is not guaranteed that all neighbour nodes will be awake to receive the measurement. Hence the requirement for optional parallel behaviour.

In the system definition of Equation 4.22, the process P can only block for its *comm* event after it has sensed some phenomenon. The process Q can either choose to block on its *comm_q*

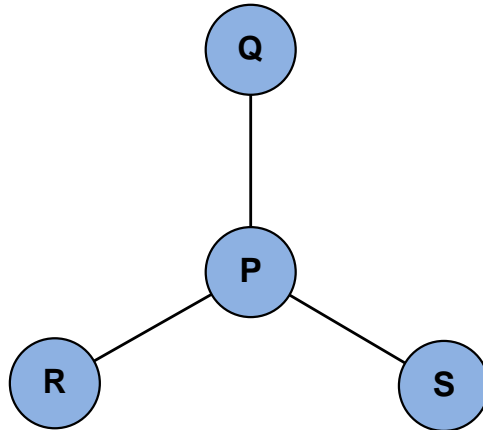


Figure 4.5: Cluster topology with processes P , Q , R and S .

event or to $sleep_q$. This decision is made internally by process Q , independent if an event is offered by the environment, hence the internal choice operator. Process Q is allowed to sleep for an indefinite period, which could simulate a node failure. Process P will be able to have its $comm$ event, because all the channel artefacts are by definition allowed to synchronise jointly on the $comm$ event. Now, depending on which event is offered by process Q , CHQ is forced into an external decision on which event to perform. If process Q is ready for $comm_q$, it will synchronise with channel artefact CHQ on $comm_q$ and communication effectively took place from process P to process Q . If process Q decided to sleep, CHQ is forced to perform a hidden internal $drop$ event, dropping the message from process P and waiting for the next message. The exact behaviour is defined for processes R and S and their respective channel artefacts CHR and CHS . This definition allows a neighbour process to be busy with something like sleeping, while the other processes are allowed to synchronise on the synchronisation event from the cluster head P .

$$P := \textit{sense}_p \rightarrow \textit{comm} \rightarrow P$$

$$Q := (\textit{comm}_q \rightarrow \textit{work}_q \rightarrow Q) \sqcap (\textit{sleep}_q \rightarrow Q)$$

$$R := (\textit{comm}_r \rightarrow \textit{work}_r \rightarrow R) \sqcap (\textit{sleep}_r \rightarrow R)$$

$$S := (\textit{comm}_s \rightarrow \textit{work}_s \rightarrow S) \sqcap (\textit{sleep}_s \rightarrow S)$$

$$CHQ := \textit{comm} \rightarrow ((\textit{comm}_q \rightarrow CHQ) \sqcap CHQ)$$

$$CHR := \textit{comm} \rightarrow ((\textit{comm}_r \rightarrow CHR) \sqcap CHR)$$

$$CHS := \textit{comm} \rightarrow ((\textit{comm}_s \rightarrow CHS) \sqcap CHS)$$

$$Q' := \left(Q \parallel_{\{\textit{comm}_q\}} CHQ \right)$$

$$R' := \left(R \parallel_{\{\textit{comm}_r\}} CHR \right)$$

$$S' := \left(S \parallel_{\{\textit{comm}_s\}} CHS \right)$$

$$\begin{aligned}
 WSN &:= P \parallel_{\{\textit{comm}\}} \left(Q' \parallel_{\{\textit{comm}\}} R' \parallel_{\{\textit{comm}\}} S' \right) \\
 &:= P \parallel_{\{\textit{comm}\}} \left(Q \parallel_{\{\textit{comm}\}} \tilde{\parallel} R \parallel_{\{\textit{comm}\}} \tilde{\parallel} S \right)
 \end{aligned} \tag{4.22}$$

The channel definitions CHQ , CHR and CHS of Equation 4.22 differ from the introduction of the channel artefact in Equation 4.8. This is because the channel definitions of Equation 4.22 only allows incoming communication events from the cluster head. The channels restrict

the neighbour nodes to synchronise with each other.

Expanding on the notion of the channel artefacts orchestrating the direction of communication, the channel artefact definition of Equation 4.8 can vary depending on the type of synchronisation required. Broadcasting, simplex, half-duplex and full-duplex synchronisation can be defined. The atomic channel artefacts required for the different types of synchronisation are given in Equations 4.23 and 4.24. The meaning of the *internal* and *external* events are the same as defined in §4.2.1.1.

$$C_{transmit} := internal \rightarrow ((external \rightarrow C_{transmit}) \square C_{transmit}) \quad (4.23)$$

$$C_{receive} := external \rightarrow ((internal \rightarrow C_{receive}) \square C_{receive}) \quad (4.24)$$

The various synchronisation types mentioned can be constructed by using these atomic channel artefacts by combining them or using them as is. It is required to know when a specific atomic channel artefact is to be used. This requires a new notation to indicate the direction of synchronisation for each synchronisation event. This notation should not be confused with the channel reading and writing definitions of [2], which are indicated with the symbols "?" and "!" respectively. The CSP channel reading and writing notation still requires that synchronisation occur jointly between processes, a restriction that optional parallelism resolves. The new directional synchronisation notation is detailed in the next subsection.

4.3.1 Directional notation

When defining systems with optional parallelism, the direction of synchronisation needs to be indicated. Instead of creating a different optional parallel operator for each of the synchronisation types, it is rather contained in the notation of the synchronisation events, and can therefore be mixed within definitions. The notation allows for arrows to be placed above an optionally synchronised event to indicate its synchronisation direction. This notation does not change the synchronisation events into different CSP symbols, but are rather just added to guide the translation to select the appropriate channel artefacts. From a pure CSP perspective, these events should be considered as the same events, and the directional arrows above them as a guide to select the appropriate channel artefacts for the type of synchronisation.

In process definitions of optionally synchronising processes, a right arrow is placed above the events which are *initiating* synchronisation events. When a process definition should *respond* to some other process initiating a synchronisation with it, a left arrow is placed above the event. This is shown in Equation 4.25 below:

$$\begin{aligned}
 WSN &:= P \quad \overset{\sim}{\parallel}_{\{comm\}} \quad Q, \text{ where} \\
 P &:= \overrightarrow{comm} \rightarrow work_p \rightarrow P \\
 Q &:= \overleftarrow{comm} \rightarrow work_q \rightarrow Q
 \end{aligned} \tag{4.25}$$

Equation 4.25 defines a process P which is allowed to initiate the *comm* event and a process Q which is only allowed to respond to such a synchronisation if it can. It is not allowed to initiate the communication back to P . Process P will therefore be modelled with a transmit channel artefact as defined in Equation 4.23 and process Q with a receive channel artefact of Equation 4.24.

For half-duplex synchronisation, the notation changes to a double arrow above the synchronisation event, which represents the bidirectional synchronisation. An example using half-duplex synchronisation is given in Equation 4.26.

$$\begin{aligned}
 WSN &:= P \quad \overset{\sim}{\parallel\!\!\parallel}_{\{comm\}} \quad Q, \text{ where} \\
 P &:= \overleftrightarrow{comm} \rightarrow work_p \rightarrow P \\
 Q &:= \overleftrightarrow{comm} \rightarrow work_q \rightarrow Q
 \end{aligned} \tag{4.26}$$

Equation 4.26 represents two processes P and Q which are each allowed to initiate a synchronisation event. The only restriction is that when a process initiates an event, it will be completed before the other process can start the synchronisation event. This is handled by the half-duplex channel artefact which will be detailed in §4.3.4.

For full-duplex synchronisation, two double arrows are placed above the synchronisation

event. An example is given in Equation 4.27.

$$\begin{aligned}
 WSN &:= P \parallel_{\{comm\}} Q, \text{ where} \\
 P &:= \overleftrightarrow{comm} \rightarrow work_p \rightarrow P \\
 Q &:= \overleftrightarrow{comm} \rightarrow work_q \rightarrow Q
 \end{aligned} \tag{4.27}$$

In this example, each process can initiate a synchronisation, even if one is already in progress. The details of the channel artefact used to allow full-duplex synchronisation is given in §4.3.5.

It should be noted that the directional notation is only used to indicate the type of channel artefact to be used for the optional parallel translation. The events are still the same from a CSP *alphabet* point of view.

4.3.2 Broadcasting

Broadcasting synchronisation is made up of processes with transmitting and receiving channel artefacts. The broadcaster will be paired with a transmitting channel of Equation 4.23 and all its neighbour nodes paired with the receiving channel artefact of Equation 4.24. This allows one process to initiate a synchronisation with its neighbours, but the neighbours are not allowed to respond using the same synchronisation event. The result is that there is directional synchronisation from the broadcaster to its neighbouring nodes. Figures 4.6a and 4.6b show how the $C_{transmit}$ and $C_{receive}$ channel artefacts will be inserted into a broadcasting example where process P is the broadcaster and processes Q , R , and S its neighbour nodes. More broadcasting examples will be covered in Chapter 6.

4.3.3 Simplex

Simplex synchronisation works on exactly the same principle as broadcasting, but only between two processes. It defines directional synchronisation only between two nodes where there is one transmitter process, using a channel artefact of Equation 4.23 and one receiver process paired with a channel artefact of Equation 4.24. Figures 4.7a and 4.7b show a simplex synchronisation example. Synchronisation is only allowed from process P as the initiator to

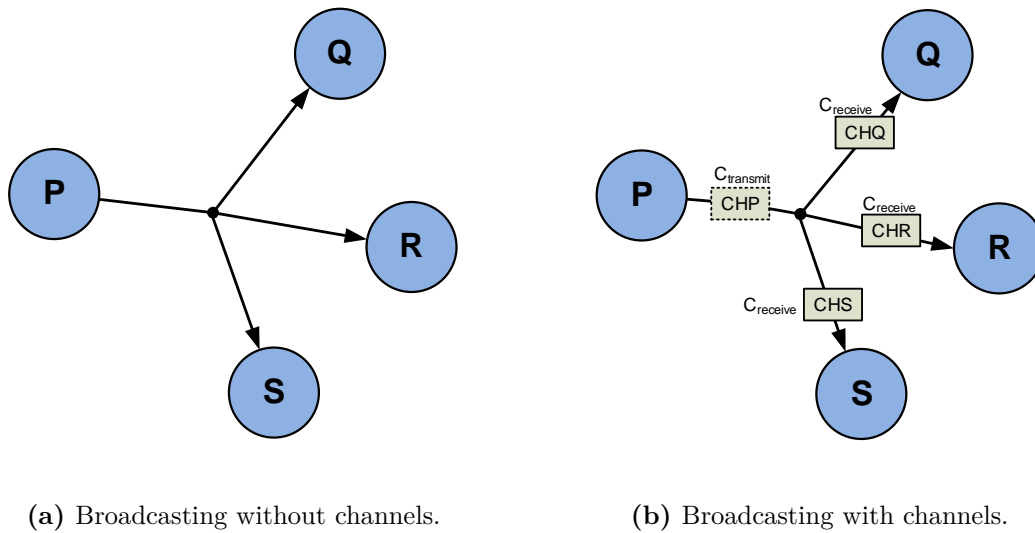


Figure 4.6: Broadcasting example showing before and after channel insertion.

process Q as the receiver. Chapter 6 details scenarios using multiple simplex channels between different nodes.

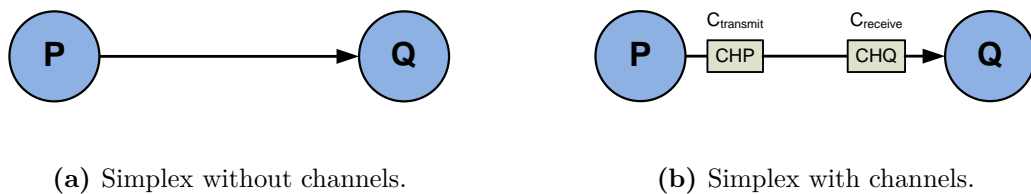


Figure 4.7: Simplex example showing before and after channel insertion.

4.3.4 Half-duplex

Half-duplex synchronisation is denoted with a double arrow above the synchronisation event which indicates bidirectional synchronisation. It is established by combining the two atomic channel artefacts of Equations 4.23 and 4.24 with the use of the external choice (\square) operator of CSP to form one half-duplex channel artefact. The use of the external choice operator means that if one side of the operator is chosen as an execution path, it is completed before a choice

can be made again on which side the execution will occur, hence half-duplex synchronisation. The half-duplex channel definition is given in Equation 4.28. Note that this equation is the same as the one defined in Equation 4.8.

$$\begin{aligned}
 C_{half-duplex} &:= C_{transmit} \square C_{receive} \\
 &:= (internal \rightarrow ((external \rightarrow C_{transmit}) \square C_{transmit})) \\
 &\quad \square (external \rightarrow ((internal \rightarrow C_{receive}) \square C_{receive}))
 \end{aligned} \tag{4.28}$$

All processes modelled with half-duplex synchronisation will have the same double arrow and there is no distinction between a transmitter and a receiver process. Half-duplex synchronisation can be between two processes or multiple processes and is typically used to model fully connected networks under optional parallelism. Figures 4.8a and 4.8b depict a simple half-duplex example where all processes are fully connected and all processes have the same $C_{half-duplex}$ channel artefact type connected to them.

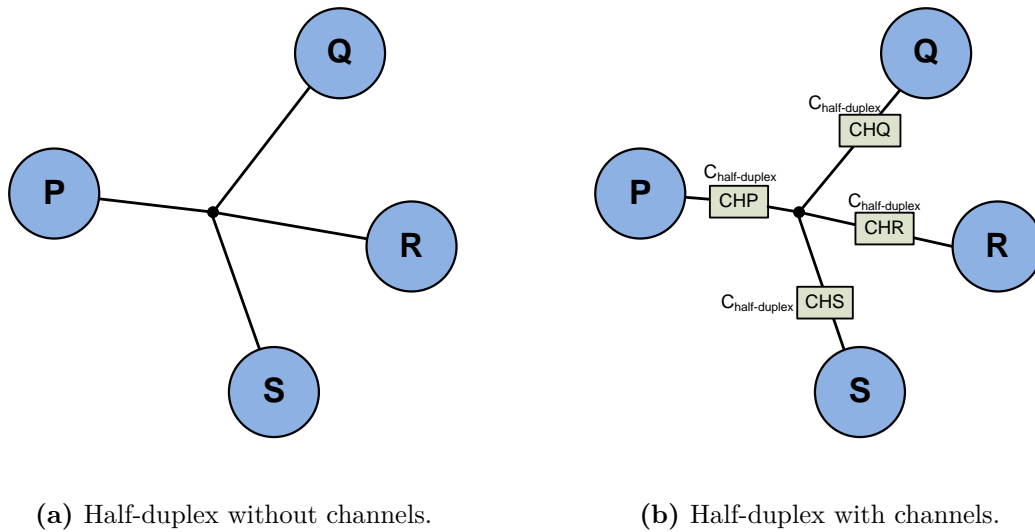


Figure 4.8: Half-duplex example showing before and after channel insertion.

4.3.5 Full-duplex

Full-duplex synchronisation share the same principles of half-duplex synchronisation, only with a different operator being used to combine the two directional channel artefacts of Equations 4.23 and 4.24 with each other. By using the interleaving operator (\parallel) of CSP, synchronisation is allowed from both ends independently. The full-duplex channel artefact is given in Equation 4.29.

$$\begin{aligned}
 C_{full-duplex} &:= C_{transmit} \parallel C_{receive} \\
 &:= (internal \rightarrow ((external \rightarrow C_{transmit}) \square C_{transmit})) \\
 &\quad \parallel (external \rightarrow ((internal \rightarrow C_{receive}) \square C_{receive})) \quad (4.29)
 \end{aligned}$$

Figures 4.9a and 4.9b show a full-duplex example. It is identical to the half-duplex example, except for the channel artefact types used, which in this case is $C_{full-duplex}$.

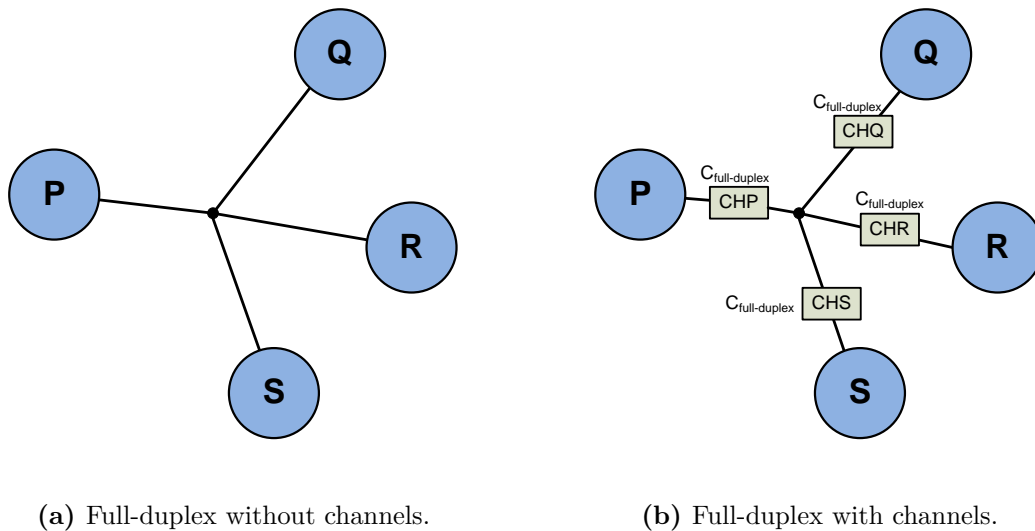


Figure 4.9: Full-duplex example showing before and after channel insertion.

4.3.6 Notation and synchronisation summary

The different synchronisation constructs which can be used to model processes with optional parallelism and their corresponding channel definitions are summarised in Table 4.1.

Synchronisation	Channel Definition	Event Notation
Broadcast	$C_{transmit}$ and $C_{receive}$	\overrightarrow{event} and \overleftarrow{event}
Simplex	$C_{transmit}$ and $C_{receive}$	\overrightarrow{event} and \overleftarrow{event}
Half-duplex	$C_{half-duplex}$	$\overleftrightarrow{event}$
Full-duplex	$C_{full-duplex}$	$\overleftrightarrow{event}$

Table 4.1: Summary of synchronisation and channel definitions.

4.4 OTHER APPROACHES CONSIDERED

Different approaches of the translational model for optional parallelism have been considered. These approaches were added here for completeness, but could also be regarded as future work.

4.4.1 Using synchronisation event artefacts

A different approach to develop translational semantics for optional parallelism using classical CSP operators is to add artificial synchronisation events to each parallel connected node process of a hypergraph of nodes. This is done for each set of node processes connected with the same hyper-edge. From Figure 4.1, this is the edge between processes P , Q and R , and is renamed from $\{comm\}$ to $\{d\}$ for this explanation. A synchronisation event is added between each process and its connected neighbour processes. This is illustrated in Figure 4.10 with the addition of the $\{a, b, c\}$ events. The WSN is then defined with the function for *synchronising events* as $WSN_{se}(N, M)$, where N denotes the number of nodes under optional parallelism and M the number of nodes which are allowed to opt out of synchronisation. The value of M implicitly defines the number and connections of the additional synchronisation events to be added. This approach has the restriction that the minimum number of processes under

optional parallelism be $N = 3$ and the maximum number of processes allowed to opt out is $M = N - 2$. The CSP representation of figure 4.10, using alphabetised parallel composition, is given in Equation 4.30.

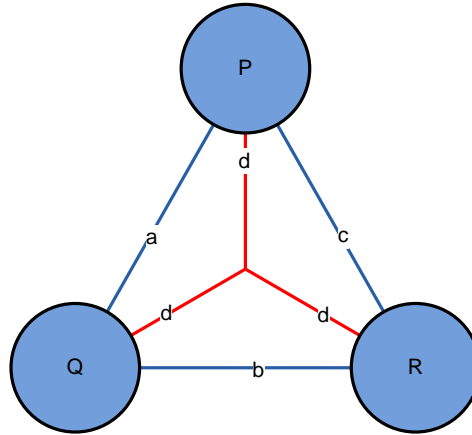


Figure 4.10: Example of 3-process hypergraph with additional synchronisation events added.

$$WSN_{se}(3, 1) := (P \alpha_P \parallel \alpha_Q Q) \alpha_{P \cup \alpha_Q} \parallel \alpha_R R \quad (4.30)$$

$$\alpha_P := \{a, c, d\}$$

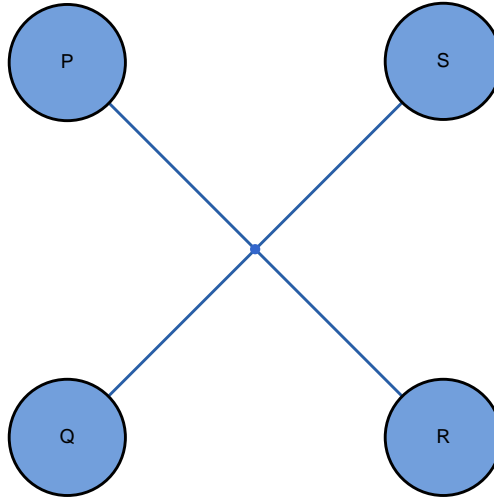
$$\alpha_Q := \{a, b, d\}$$

$$\alpha_R := \{b, c, d\}$$

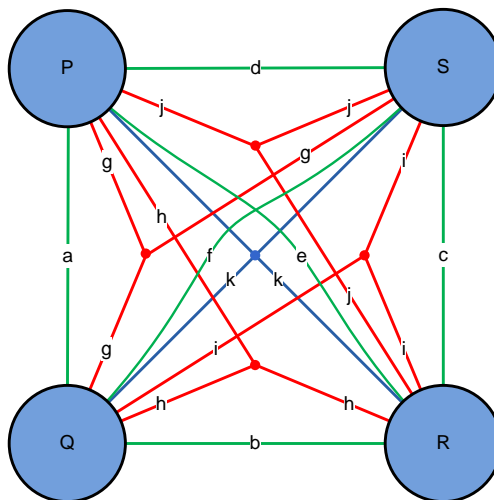
From Figure 4.10, the event $\{d\}$ corresponds to the hyper-edge, e_1 of Equation 4.14. The events $\{a, b, c\}$ are artificially added to create a synchronisation event in the case a node opts out. In other words, if process R decides to opt out, the common synchronisation $\{d\}$ is not possible and therefore $\{a\}$ will be observed.

This approach is powerful to describe exactly which process(es) opted out, but implementing the scenario becomes tedious because of the non-linear growth and overhead of the alphabets. To illustrate the effect of alphabets, consider a fully connected network graph of $N = 4$ processes as in Figure 4.11a. The extra events needed to model optional parallelism for at

most $M = 2$ processes opting out can be seen in Figure 4.11b. The effect of the extra alphabet items needed is also visible in the CSP definition of Equation 4.31.



(a) Normal hyper-edge between 4 processes.



(b) Graph with artificial synchronisation events added.

Figure 4.11: Event definitions for $N = 4$ and $M = 2$.

The edges between the processes are defined with their shared events, these are indicated in colour in Figure 4.11b.

$$\begin{aligned}
 WSN_{se}(4, 2) &:= ((P \ \alpha P \parallel_{\alpha Q} Q) \ \alpha P \cup_{\alpha Q} \parallel_{\alpha R} R) \ \alpha P \cup_{\alpha Q} \cup_{\alpha R} \parallel_{\alpha S} S & (4.31) \\
 \alpha P &:= \{a, d, e, g, h, j, k\} \\
 \alpha Q &:= \{a, b, f, g, h, i, k\} \\
 \alpha R &:= \{b, c, e, h, i, j, k\} \\
 \alpha S &:= \{c, d, f, g, i, j, k\}
 \end{aligned}$$

From a modelling perspective, this approach is difficult because each process will need to be ready to communicate any of its events as one process will have no *a priori* knowledge of which neighbouring process will opt out at the next step. This has an enormous effect on the number of possible states, as every process will need to cater for all possible synchronisation combinations, which increases non-linearly with the addition of a single process to the network. This approach has not been further investigated as it seemed fruitless based on too generic results of which the main contributing one is that all processes can perform any combination of all events at any time. This has infinite, diverging traces for each process, which falls out of the scope of the problem being addressed in this dissertation.

4.4.2 Using stochastic CSP

The opting out of processes could be stochastically defined by incorporating the rate of failure of WSNs. This failure rate can be defined as any rate at which a process will stop engaging in communication, because of resource depletion, component failure or communication failure. While specifications are refined into actual implementations, reliability and design trade-offs can be stochastically defined. This is implemented with an internal choice operator of the channel artefact of Equation 4.8, as is shown in Equation 4.32. The internal choice operator is used because the choice of opting out is now defined by the channel artefact and not on the availability of a synchronisation event from the node process.

$$\begin{aligned}
 C &:= (internal \rightarrow ((external \rightarrow C) \sqcap_p C)) \\
 &\quad \square (external \rightarrow ((intenal \rightarrow C) \sqcap_p C)) & (4.32)
 \end{aligned}$$

The stochastic translation defines a failure rate or rate of *opting out* for each process. A value of $p = 0$ means that the probability of a node opting out is 0, which means that the process will

always synchronise on common events. This translates to general parallelism in CSP. A value of $p = 1$ means that the process will always opt out, meaning it will never synchronise and by the deadlock free definition, will never block to synchronise. This translates to interleaving in CSP. With a failure rate $0 < p < 1$, the processes are under optional parallel behaviour. The general translation of optional parallelism of Equation 4.21 then includes a failure rate parameter as is shown in Equation 4.33

$$\tilde{\parallel}_{i=1}^n (N_i, \alpha N_i, p_i) := \begin{cases} \left\{ \begin{array}{l} \parallel_{X, i=1}^n (N_i, \alpha N_i) & \text{if } p_i = 0 \forall 1 \leq i \leq n \\ X = \bigcup_{i=1}^n \alpha N_i \end{array} \right. \\ \parallel_{i=1}^n (N_i, \alpha N_i) & \text{if } p_i = 1 \forall 1 \leq i \leq n \\ \left(\parallel_{i=1}^n N_i \right) \parallel_Y \left(\parallel_{j=1}^c (C_j, \alpha C_j, p_i) \right) & \text{otherwise} \\ Y = \left(\bigcup_{i=1}^n \alpha N_i \right) \cap \left(\bigcup_{j=1}^c \alpha C_j \right) \end{cases} \quad (4.33)$$

WSN system models can be more realistically modelled to represent real-world WSN behaviour.

CHAPTER 5

SOFTWARE TOOLS

The CSP model-checker, ProCSP, can be used to check WSN systems for possible deadlocks and if a specific trace can be generated by a WSN system definition, i.e. trace refinement. Although ProCSP is only a research tool, its capabilities are sufficient to check the models and trace refinements in concept of this dissertation. An additional tool is needed to convert the CSP definitions containing the optional parallel operator to definitions containing only classical CSP operators for ProCSP to be used. This is due to optional parallelism not being implemented in current model-checkers yet. These conversions are suitable as input to CSP model-checkers. A final tool is needed to check for trace refinement between $OptPar^T$, using classical CSP operators and the channel artefact, and $OptPar$ as per [17]. Traces of systems communicating with $OptPar$ are computed with a custom software implementation and a model file is generated with the $OptPar^T$ translation. The model file is a suitable input for ProCSP for deadlock analysis and to check for trace refinement.

5.1 OPTIONAL PARALLEL TO CSP DEFINITION GENERATOR

The optional parallel to CSP definition generator, OptoCSP¹, is used to convert CSP definitions containing $OptPar$ to CSP definitions containing only classical CSP operators ($OptPar^T$). This is done by remodelling the process definitions and with the addition of the channel artefact of Equation 4.8. OptoCSP is a new software tool developed to aid in the conversion from CSP definitions containing $OptPar$ to systems containing $OptPar^T$, while still retaining the same behavioural characteristics in the traces domain \mathcal{T} .

¹Source can be found at <https://github.com/theunssteyn/OptoCSP>

5.1.1 Requirements

OptoCSP receives a file with the process and system definitions as input. The format of the file has the requirements listed below.

1. All process definitions shall be defined under the "PROCESS:" heading.
2. The system definition shall be defined under the "SYSTEM:" heading.
3. All process and system definitions shall have space delimited operators and operands.
4. All process and system definitions shall be defined in a single line of text.

An example of an input file is shown in Listing 5.1. Note that the CSP_M syntax for the $OptPar^T$ operator $\underset{X}{\parallel}$ is given as $[|||X|||]$, where X denotes the synchronisation set.

```

PROCESS:
P = meas_p -> comm -> P
Q = meas_q -> comm -> Q

SYSTEM:
WSN = P [|||comm|||] Q
  
```

Listing 5.1: Example input file ready for conversion from $OptPar$ to $OptPar^T$.

The user interface of OptoCSP shall:

1. provide an output of the input file contents in a text editor;
2. provide the translated output in a text editor; and
3. be able to save the translated output.

The conversion engine of OptoCSP shall:

1. check the syntax of the given input;
2. provide output to the user in the case of an error; and

3. convert systems to classical CSP, based on the input definition.

5.1.2 System overview

The block diagram of the conversion engine of OptoCSP is given below in Figure 5.1.

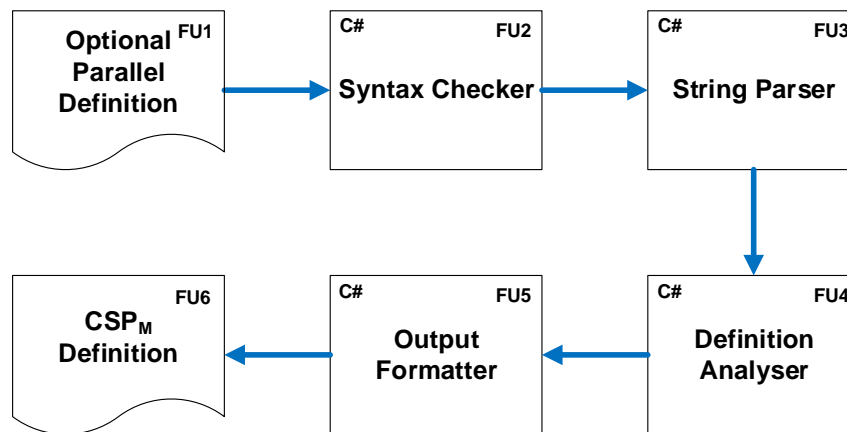


Figure 5.1: Functional block diagram of OptoCSP.

Functional Unit (FU) 1 represents the input file of Listing 5.1. This file contains all the process and system definitions which need to be translated from an optional parallel definition to a classical CSP definition. FU2, the Syntax Checker, is responsible for checking the input file for syntax errors. If an error is detected, the conversion process is aborted. The Syntax Checker also has the responsibility for providing details regarding the particular syntax error, if one is found. FU3 parses the **PROCESS** and **SYSTEM** sections from strings into internal **Process** and **System** objects. Each **Process** object has an internal data structure containing a sequential list of operator/operand objects. This internal list represents each process' definition as given in the input file FU1. The **System** object has an internal data structure containing a sequential list of operators and processes. FU4 analyses the **System** object to detect which processes are operating under optional parallelism. It then generates new process definitions with their synchronisation events renamed to internal unique identifiers. Finally, a **Channel** object is added to the process definitions for each synchronisation event that has been renamed. A **Channel** object has a fixed internal process definition as per Equation 4.8, with only its events renamed to correspond to the renamed synchronisation event of the process it is representing, and the external synchronisation event defined in the **System**

object. FU5 is responsible for formatting the **Process**, **Channel** and **System** objects into a CSP system definition using only existing classical CSP operators. This FU generates the CSP output file, FU6, which can be used in model-checking tools like ProCSP.

5.1.3 Implementation

OptoCSP is a Microsoft Windows Forms application, implemented in C# and using the .Net Framework 4.5. The user interface is event-driven, with the user given the option to open a file for input, do the conversion and provide a file name for the output. The input and output are also presented next to each other in text format to the user. A screen capture of the OptoCSP tool is given in Figure 5.2.

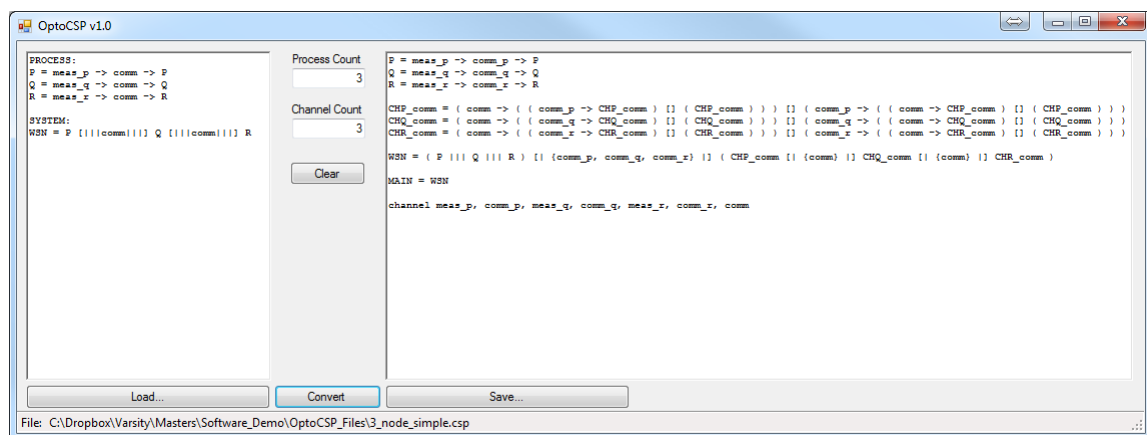


Figure 5.2: Screen capture of the user interface of OptoCSP.

The FU's of Figure 5.1 are implemented using an object oriented approach, making the modules reusable for more advanced future implementations.

5.1.4 Test and validation

The output of the OptoCSP translation was checked and scrutinised by hand. The examples of Chapter 6 were given as input and each of the output model files were checked with the expected result. The output model files were opened in ProCSP, which checked for CSP syntax. After a successful syntax check, the model was marked as executable by ProCSP. Specific system verifications like trace refinement and deadlock freedom are left for a later stage of system analysis. The concern here is to test if the OptoCSP tool can correctly convert

a CSP definition using the optional parallel operator, to a definition containing only classical CSP operators.

Table 5.1 gives a summary of the equations used to test the new OptoCSP tool. It was decided to present the test results here in order to keep the details regarding the OptoCSP tool together and because this is not the main focus of this dissertation. Each translated equation has been validated against the output generated by a different tool, OpTrace, which will be detailed in the next section. The model outputs were generated by the OpTrace tool, given the adjacency lists defined in Chapter 6 for each of the equations tested. If the translation of an equation using OptoCSP had the same CSP output as the model generated by OpTrace, given its adjacency list, the test passed.

Equation	Model Reference	Pass	Equation	Model Reference	Pass
6.2	B.1.1.1	✓	6.27	B.3.1.3	✓
6.7	B.1.1.2	✓	6.29	B.3.2.1	✓
6.9	B.1.1.3	✓	6.31	B.3.2.2	✓
6.11	B.1.2.1	✓	6.33	B.3.2.3	✓
6.13	B.1.2.2	✓	6.35	B.3.3.1	✓
6.15	B.1.2.3	✓	6.37	B.3.3.2	✓
6.17	B.2.1.1	✓	6.39	B.3.3.3	✓
6.19	B.2.1.2	✓	6.41	B.4.1.1	✓
6.21	B.2.1.3	✓	6.43	B.4.1.2	✓
6.23	B.3.1.1	✓	6.45	B.4.1.3	✓
6.25	B.3.1.2	✓			

Table 5.1: OptoCSP test results.

From Table 5.1, it is clear that all of the conversions were syntactically valid. For the given applicability of OptoCSP, it was decided that these tests were sufficient. Formal unit testing and software qualification seemed to fall outside of the scope of this dissertation. If OptoCSP could handle the examples of this dissertation, it was deemed error free and safe to use in this dissertation. It can therefore be concluded that OptoCSP can be used for the equations with the same structure as the ones tested in this dissertation.

5.2 AUTOMATED TRACE VERIFICATION

With optional parallelism being a new addition to the CSP language, there is currently no CSP tool implementing it. The implementation of optional parallelism needs to follow a different approach. The translation of Equation 4.21 can be implemented in CSP_M for a given WSN scenario. This allows model properties to be checked such as deadlock, livelock, divergence and safety. Other properties such as trace refinement and failure divergence refinement can also be checked by CSP model-checkers, given that the models have a trace input generated from the step law of optional parallelism of Equation 3.32.

As a first model-checking test, the system should be deadlock free. Then it needs to be confirmed if the two approaches to optional parallelism, the first approach being $OptPar$ and the second approach being $OptPar^T$, has a trace refinement relation for a given WSN scenario. In the case where a refinement relation between the two approaches is found, it can be concluded only that a trace refinement relation exists for that specific WSN scenario. Proving this for *all* possible WSN scenarios is left for future work. In other words, to perform this scenario-based verification, the traces of $OptPar$ need to be compared against the traces of $OptPar^T$ for a specific WSN scenario to check if a refinement relation exists.

The step law of $OptPar$ (Equation 3.32) has been implemented with a recursive routine to compute the traces between processes. The implementation has been tediously scrutinised by hand. The trace output of the program can then be used in a model-checking tool for refinement checks.

Deadlock freedom of the translation has been confirmed with ProCSP, as was expected due to the design principles used in §4.2.2. The trace refinement check was used to see if the specification is refined by its implementation:

$$SPEC \sqsubseteq_T IMPL \tag{5.1}$$

Where $SPEC$ is $OptPar^T$ of Equation 4.21 and $IMPL$ the trace output of OpTrace, based on $OptPar$ [17].

To determine if $OptPar^T$ has a refinement relationship with $OptPar$, *all* the possible traces for the given scenario should be checked for trace refinement. ProCSP provides such a function where it can be checked if a trace sequence can be generated by the system definition. Listing

5.2 shows how this is done for the trace set $\langle meas_p, comm, meas_q, meas_p \rangle$, using a system definition $WSN_{P,Q}$.

```

trace = meas_p -> comm -> meas_q -> meas_p -> STOP
assert WSN_P_Q [T= trace

```

Listing 5.2: CSP_M syntax for trace refinement.

If all the possible traces of $OptPar$ (\uparrow_X of [17]) are checked against $OptPar^T$ ($\tilde{\parallel}_X$ of Chapter 4), as shown in listing 5.2, and all the traces are confirmed to refine $OptPar^T$, it can be deduced that the specific optional parallel translation is refined by its implementation using $OptPar$. To test the other way around, it is needed that ProCSP implements $OptPar$ for system analysis on network definitions with processes communicating under optional parallelism. This is currently not the case and is left for future work. It can therefore currently only be checked if $OptPar$ refines $OptPar^T$, for a given WSN system definition, in the trace domain \mathcal{T} .

Trace verification of different systems is a cumbersome task as the traces of processes communicating under optional parallelism have to be computed from first principles, using the step laws of [17], which are also given in Equation 3.32. A trace generator for $OptPar$ is therefore needed, which formats the result into CSP_M such that a model-checker is able to decode the result as input. This functionality was implemented in the OpTrace² tool.

5.2.1 Requirements

The following list details the software requirements of the OpTrace tool. OpTrace shall:

1. receive a file with the adjacency list of a network definition with the format given in Listing 6.1;
2. compute all the possible traces between the processes under optional parallelism, using the step laws of Equation 3.32;
3. reuse the OptoCSP translation engine to generate a CSP system definition of the input adjacency list;

²Source can be found at <https://github.com/theunstejn/OpTrace>

4. combine the computed traces with the translated CSP system definition into a single CSP_M output file;
5. call the ProCSP module with the output file for trace refinement checking; and
6. provide output to the user containing the results.

5.2.2 System overview

OpTrace is automated as far as possible for trace refinement checks. It accepts an adjacency list, representing WSN topological definition as input. The node process and channel definitions are fixed and are therefore added as per the topological definition of the WSN, one node process for each WSN node and one channel for each communication link between two nodes. The traces of the node processes are generated with the *OptPar* implementation. A CSP_M *OptPar*^T system definition, containing process- and channel definitions, is generated based on the input topological information. A CSP model file is generated which contains the CSP_M statements required to check for trace refinement, using the traces computed by the *OptPar* implementation. ProCSP is invoked from OpTrace with the model file as input to check if the computed traces refine the CSP_M model. In Figure 5.3, the functional block diagram of OpTrace, FU1 represents the input adjacency list as a text file with the format given as discussed in §5.2.3. The adjacency list defines the topology of the WSN. FU2 represents the WSN Definition Generator. With the given adjacency list (FU1), an internal set of node- and channel processes is generated. The channel processes are linked to their corresponding node processes with the use of the adjacency list definition. If a duplicate channel is encountered, the previous definition will be re-used. Internally, the node processes and channels are connected as per Figure 4.4. This FU is implemented in C#. The set of node and channel processes is given as input to FU3, FU4 and FU5.

FU3 represents the Trace Generator, implemented in C#. This FU uses the *OptPar* implementation discussed later in §5.2.4.2 to compute all the possible traces between the node processes. The finite traces are computed with each process executing all of its possible traces (in one cycle of execution before repeating a trace) and computing how these possible traces interact with each other under optional parallelism. These traces are kept in expanded form to be included in the CSP model file (FU6) for trace refinement checks. The channel

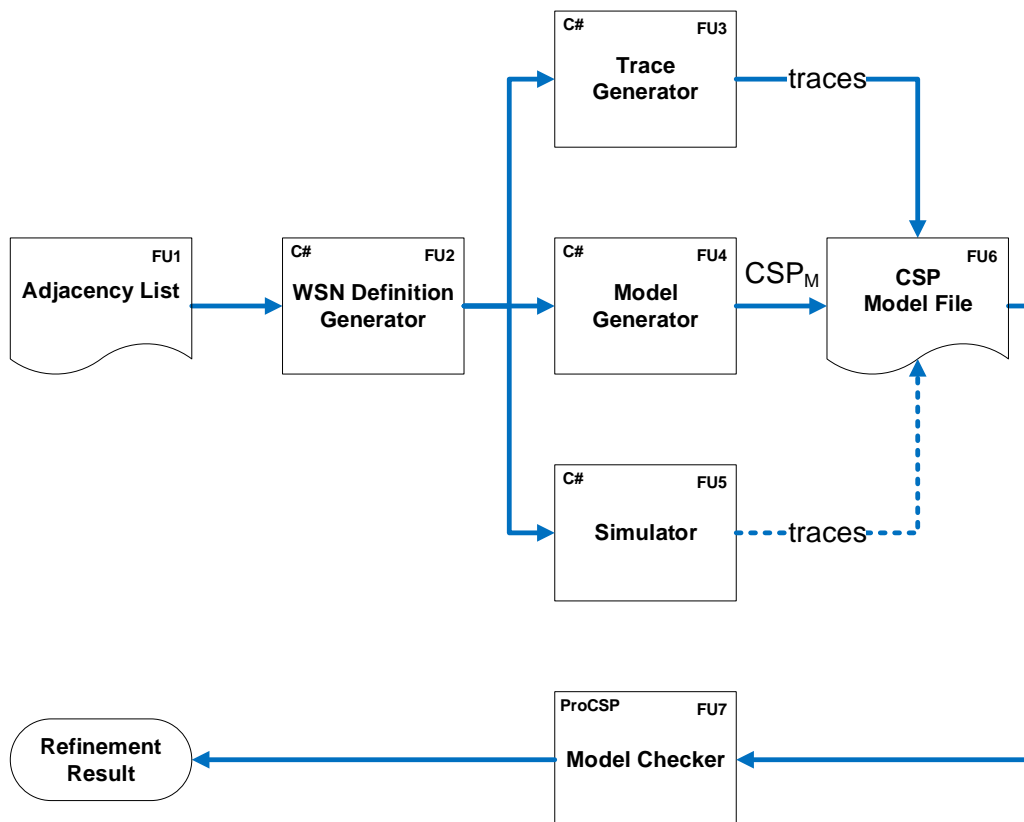


Figure 5.3: Functional block diagram of OpTrace.

processes are not used in this FU because, as stated previously, it serves only as a modelling artefact for the translation of optional parallelism to classical CSP. The step laws of Equation 3.32 are defined in this FU.

The Model Generator, represented by FU4, is used to generate a CSP_M system definition based on the list of node- and channel processes. The system definitions are generated by using the process definitions of Chapter 6 and $OptPar^T$, according to the channel definition of Equation 4.8. The definition is formatted so that ProCSP accepts it as a valid CSP input file. This FU is implemented in C#.

The Simulator (FU5) simulates the communication between the node processes by instantiating multiple software processes, each running on its own CPU thread, representing the WSN node and channel processes. The node processes are implemented as per their definitions in Chapter 6 and the channel processes as per Equation 4.8. These instantiations interact with each other randomly and their interaction, represented as traces, are recorded. A simulation run represents only one path of continuous execution of all the trace events of the nodes and channels. The recorded traces serve as the trace input to FU6 for trace refinement verification. More detail on the implementation of this FU is given in §5.2.5. This FU is implemented in C#.

FU6 represents the model file. It comprises of CSP_M process definitions, a system definition and a set of possible traces to do trace refinement checking on. The traces are either given as input from FU3, the theoretical trace generator, or FU5, the simulated trace generator.

The Model Checker, ProCSP, given as FU7, is used to check the CSP assertions in the model file (FU6). It is invoked by OpTrace with a model file (FU6) as input. The results of ProCSP are checked to see if all assertions pass the trace refinement check. If all assertions succeeds the trace refinement check, it is concluded that $OptPar^T$ is refined by $OptPar$, for the specific input WSN.

OpTrace is a Microsoft Windows Forms application, implemented in C# and using the .Net Framework 4.5. The user interface is event-driven, with the user given the option to load a network adjacency list, generate the traces of the loaded network, build a CSP model-file and invoke the ProCSP model-checker to check for trace refinement. The computed traces are given as output and the results of the trace refinement checks are output to screen. A

screen capture of the OpTrace tool is given in Figure 5.4.

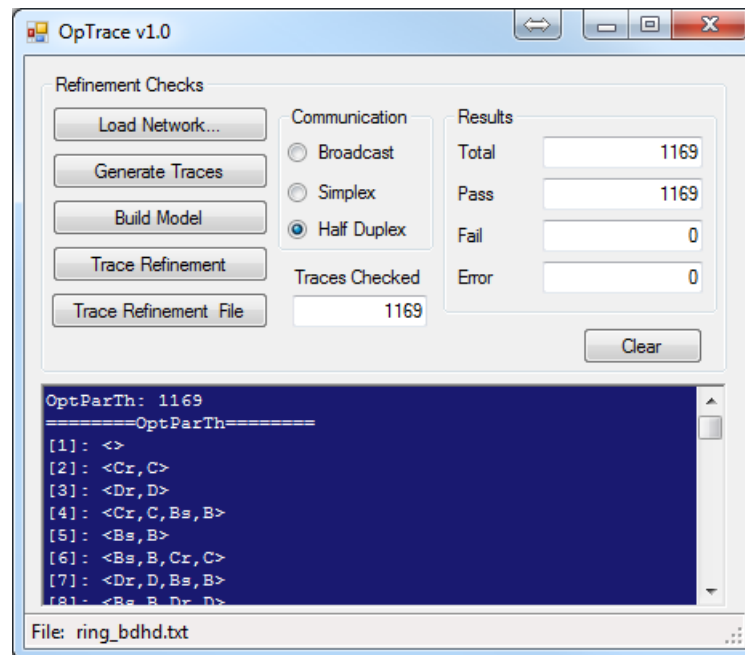


Figure 5.4: Screen capture of the user interface of OpTrace.

As with OptoCSP, the FU's of OpTrace given in Figure 5.3 are implemented using an object oriented approach, making the modules reusable for more advanced future implementations.

5.2.3 Adjacency list notation

Adjacency lists [110] are used to define the WSN topologies. An adjacency list comprises of an array of vertex-indexed lists of the vertices adjacent to each vertex. In other words, it is a list where all the connected processes of a system are given. An *adjacency list* was chosen above an *adjacency matrix* or a *set of edges*, firstly because an adjacency matrix requires a matrix of size N^2 boolean variables where N represents the node count, and secondly because a set of edges requires a lot of processing to determine if one node is adjacent to another. An advantage of adjacency lists and adjacency matrices is that they can represent directional graphs in the case where communication takes place in one direction only.

The notation used for the adjacency lists used in this dissertation is proprietary. The types of synchronisation (discussed in §4.3) between processes need to be distinguished from the

adjacency lists. The notation makes use of brackets and braces to distinguish directional and non-directional edges. Note that a non-directional edge represents communication in both directions and effectively represents a bidirectional edge. The following rules are defined for the adjacency lists used in this dissertation.

- The first parameter in a line is always the source of an edge and all subsequent parameters in the line are the destination(s) of the edge.
- Parameters between brackets "(" and ")" indicate a directional edge.
- Parameters between braces "{" and "}" indicate a non-directional edge.

Figure 5.5 shows a network graph and its corresponding adjacency list for processes P and Q . This definition is typically found in simplex synchronisation definitions.

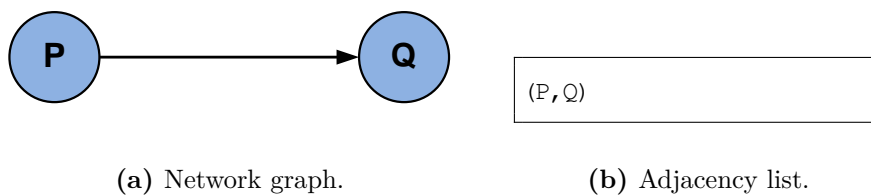
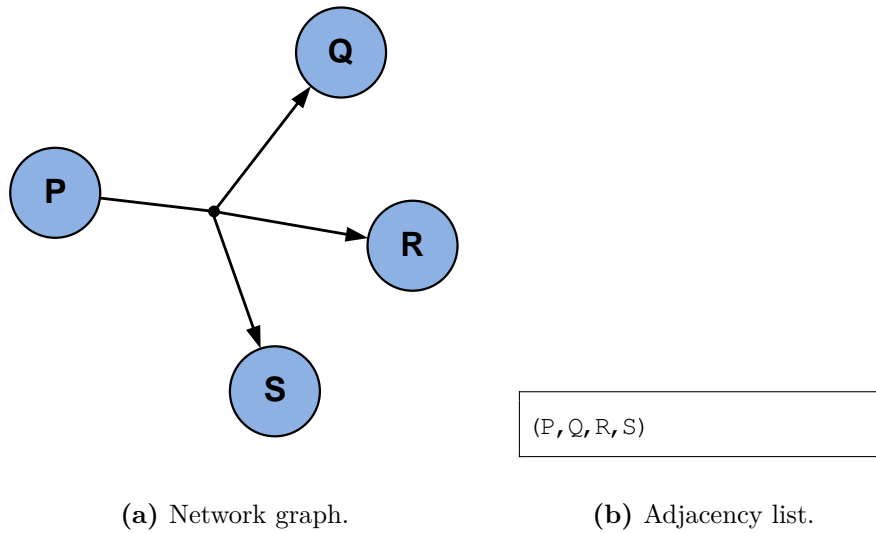


Figure 5.5: Network graph and adjacency list for directional source P to destination Q .

Figure 5.6 gives a hyper-edge system and its corresponding adjacency list with source node P and receiving nodes Q , R and S . This system is typically defined for broadcasting systems. Figure 5.7 shows a non-directional edge and its adjacency list for nodes P and Q . This example will typically be used for a point-to-point bidirectional half-duplex synchronisation. Figure 5.8 shows a network graph and its adjacency list for a non-directional hyper-edge between nodes P , Q , R and S . This example is used for bidirectional half-duplex synchronisation.

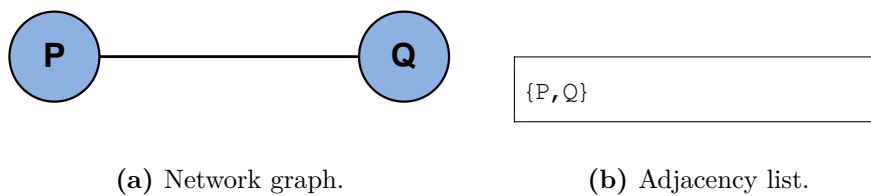
OpTrace will decode the adjacency lists as per their directional definitions and indicate to the user which synchronisation has been detected. More advanced adjacency list definitions are given for various WSN topology scenarios in Chapter 6.



(a) Network graph.

(b) Adjacency list.

Figure 5.6: Network graph and adjacency list for hyper-edge with source P to destinations Q , R and S .



(a) Network graph.

(b) Adjacency list.

Figure 5.7: Network graph and adjacency list for non-directional edge between P and Q .

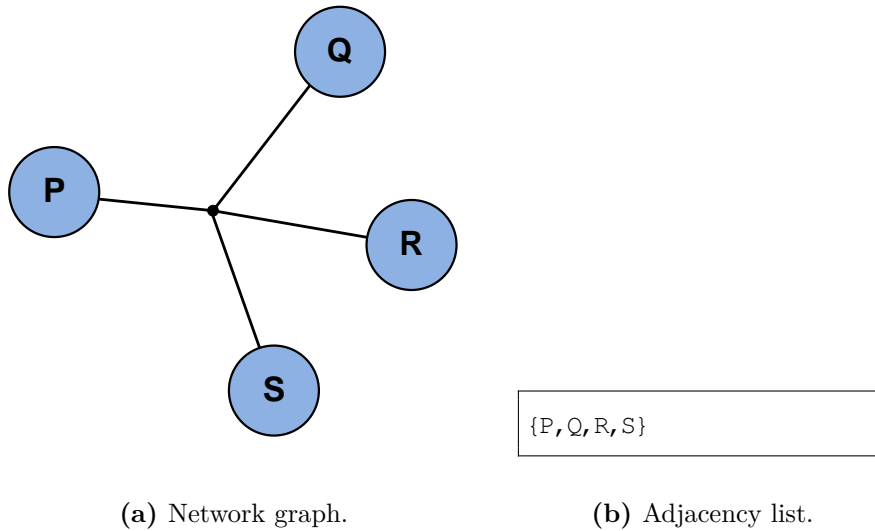


Figure 5.8: Network graph and adjacency list for non-directional hyper-edge between P , Q , R and S .

5.2.4 Theoretical trace generation

The Trace Generator (FU3 of Figure 5.3) implements both optional- as well as generalised parallelism. It generates all the possible trace combinations between two or more communicating processes, given their finite set of individual traces. It also receives a synchronisation set, which is the set of common events between processes. All processes with traces containing elements of the synchronisation set, are allowed to communicate with each other. All possible combinations of traces between the processes are computed, based on the step laws of either the generalised parallel operator of CSP [2] or *OptPar* [17]. The definitions of the step laws for the generalised and the optional parallel operators are given in Chapter 3 in Equations 3.22 and 3.32 respectively. By receiving an empty synchronisation set for either of the two implemented operators, interleaving results, which is given in Equation 5.2.

$$P \underset{\{\}}{\parallel} Q = P \underset{\{\}}{\parallel\parallel} Q = P \underset{\{\}}{\uparrow} Q \quad (5.2)$$

The implementation and checking approach followed was to first implement the operator behaviour which could be tested with another CSP tool like ProCSP. This approach had the advantage of being able to test the implementation for usability and software defects with an independent tool. With the generalised parallel operator being a commonly used CSP

operator, implemented in all credible CSP model-checkers, it was the operator of choice to be implemented first. If the implementation of the generalised parallel operator could be tested and qualified for the purpose of the scenarios presented in this dissertation, the implementation of optional parallelism is simplified by only lifting the synchronisation restrictions of the general parallel operator. This approach was followed because there are no other tools that could be used to test the implementation. The only way to check the newly developed *OptPar* routine is to do it manually by hand. The implementation of the general parallel operator function allows only for a binary process composition, but the function calls are allowed to be staggered, effectively allowing multiple processes to synchronise on the same event, i.e. n-ary synchronisation. The step laws of the general parallel operator (Equation 3.22) were implemented by means of a recursive function. The function receives a set of possible traces for two communicating processes as well as a synchronisation set. It was decided not to derive the synchronisation set from the intersection of the alphabets of the two communicating processes, to give the user more control over the synchronisation behaviour.

5.2.4.1 Generalised parallel implementation

Pseudo code of the recursive general parallel operator function is given in Algorithm 1. This serves the purpose of not being programming-language specific. The function receives two strings, s and t , representing traces as input, a synchronisation set X and a history of traces. The history will be empty when the function is called for the first time. The start of the function deals with the exit conditions of the recursive function where the one process is allowed to continue after the other one has no more traces left. The recursion ends when both trace sets have finished producing traces, or when a deadlock scenario occurs. The rest of the function deals with the step laws of general parallelism. At first, in lines 11 to 22, it is checked if $s[0]$ is contained in the synchronisation set X . If it is equal to $t[0]$, the first condition of the step law of Equation 3.22 is met. If $s[0]$ is not equal to $t[0]$, the condition where both processes are trying to communicate different events in their synchronisation set occurs, and the communication deadlocks. If $t[0]$ is not in the synchronisation set, $s[0]$ blocks for synchronisation and trace set t moves on. This is the third condition of the step law of Equation 3.22. Lines 23 to 34 explain the same behaviour, but with s and t interchanged and satisfying the fourth condition of Equation 3.22. The second condition of the step law, where either process may proceed if their events are not in the synchronisation set X , is handled in

lines 36 to 39. This concludes the implementation of the generalised parallel operator as all the conditions of the step law of Equation 3.22 are covered.

The trace semantics of the general parallel operator, given in §3.2.4.2, are encapsulated in the operator implementation of Algorithm 1. The traces of general parallelism of Equation 3.28 indicates that *all* possible combinations of the traces between two processes are to be tested with the step law and the trace semantics thereof. This is done in the parent function which calls the recursive general parallel function by an exhaustive *for* loop. Finally, all duplicate traces are removed from the set of resulting traces.

Algorithm 1 Pseudo code for general parallelism of [2].

```

1: procedure GENPARALLEL(TraceSet s, TraceSet t, Set X, TraceSet traceHistory)
2:   if s.Count == 0 and t.Count == 0 then
3:     return
4:   else if s.Count == 0 then
5:     traceHistory.Append(t[0])
6:     GENPARALLEL(s, t[1..end], X, traceHistory)
7:   else if t.Count == 0 then
8:     traceHistory.Append(s[0])
9:     GENPARALLEL(s[1..end], t, X, traceHistory)
10:  else
11:    if s[0] is contained in X then
12:      if s[0] == t[0] then /* Sync */
13:        traceHistory.Append(s[0])
14:        GENPARALLEL(s[1..end], t[1..end], X, traceHistory)
15:      else/* s needs to block */
16:        if t[0] is contained in X then /* Deadlock */
17:          traceHistory.Append("DL")
18:          return
19:        else/* s stays current, t moves on */
20:          traceHistory.Append(t[0])
21:          GENPARALLEL(s, t[1..end], X, traceHistory)
22:        end if
23:      end if

```

Algorithm 1 Pseudo code for general parallelism of [2] (continued).

```

24:     else if  $t[0]$  is contained in  $X$  then
25:         if  $s[0] == t[0]$  then /* Sync */
26:              $traceHistory.Append(t[0])$ 
27:             GENPARALLEL( $s[1..end]$ ,  $t[1..end]$ ,  $X$ ,  $traceHistory$ )
28:         else/*  $t$  needs to block */
29:             if  $s[0]$  is contained in  $X$  then /* Deadlock */
30:                  $traceHistory.Append("DL")$ 
31:             return
32:         else/*  $s$  moves on,  $t$  stays current */
33:              $traceHistory.Append(s[0])$ 
34:             GENPARALLEL( $s[1..end]$ ,  $t$ ,  $X$ ,  $traceHistory$ )
35:         end if
36:     end if
37:     else/* Split, because 2 combinations are possible */
38:          $traceHistory.Append(s[0])$ 
39:         GENPARALLEL( $s[1..end]$ ,  $t$ ,  $X$ ,  $traceHistory$ )
40:          $traceHistory.Append(t[0])$ 
41:         GENPARALLEL( $s$ ,  $t[1..end]$ ,  $X$ ,  $traceHistory$ )
42:     end if
43: end if
44: end procedure

```

5.2.4.1.1 Validation

The general parallel operator was first validated by using simple examples of two communicating processes running concurrently. This tested the binary relationship of the operator. The point-to-point example presented in Equation 6.1 was modified slightly to keep the trace set results tidy. The process definitions of Equation 6.1 were renamed to only contain the first letter of each event and another process R , with the same structure as P and Q , was added. First, the definitions of P and Q were used and later on, the definition of R was added to test the operator staggering functionality. The processes were renamed

to the following definitions:

$$P := m_p \rightarrow c \rightarrow w_p \rightarrow P \quad (5.3)$$

$$Q := m_q \rightarrow c \rightarrow w_q \rightarrow Q$$

$$R := m_r \rightarrow c \rightarrow w_r \rightarrow R$$

$$X := \alpha P \cap \alpha Q \cap \alpha R = \{c\}$$

The trace set scenario for each process contained a finite set of traces to complete one cycle of execution. All the possible traces between processes P and Q in parallel, $traces(P \parallel_X Q)$, for their finite sets of traces, were generated by the general parallel function to be:

$$\begin{aligned} &\{ \langle \rangle, \langle mp \rangle, \langle mq \rangle, \\ &\langle mp, mq \rangle, \langle mq, mp \rangle, \\ &\langle mp, mq, c \rangle, \langle mq, mp, c \rangle, \\ &\langle mp, mq, c, wp \rangle, \langle mp, mq, c, wq \rangle, \langle mq, mp, c, wp \rangle, \langle mq, mp, c, wq \rangle, \\ &\langle mp, mq, c, wp, wq \rangle, \langle mq, mp, c, wp, wq \rangle, \langle mp, mq, c, wq, wp \rangle, \langle mq, mp, c, wq, wp \rangle \} \end{aligned}$$

For $traces(P \parallel_X Q \parallel_X R)$, there are 123 trace set results, compared to 15 for $traces(P \parallel_X Q)$. The full set of traces for $P \parallel_X Q \parallel_X R$ is given in Appendix A.

It can be noted that all trace results have at most one $\{c\}$ event, indicating that the processes have synchronised on the event. Deadlocked events are marked with a **DL** keyword. Consider the process definitions of Equation 5.4 where each process operates normally, i.e. measuring, communicating and working, or it can measure something, do a protocol synchronisation and then start executing from the beginning, discarding the measurement.

$$S := (m_s \rightarrow c \rightarrow w_s \rightarrow S) \square (m_s \rightarrow p \rightarrow S) \quad (5.4)$$

$$T := (m_t \rightarrow c \rightarrow w_t \rightarrow T) \square (m_t \rightarrow p \rightarrow T)$$

$$U := (m_u \rightarrow c \rightarrow w_u \rightarrow U) \square (m_u \rightarrow p \rightarrow U)$$

$$Y := \alpha S \cap \alpha T \cap \alpha U = \{c, p\}$$

The $traces(S \parallel T)$ are given as:

$$\{ \langle \rangle, \langle mt \rangle, \langle ms \rangle$$

$$\langle ms, mt \rangle, \langle mt, ms \rangle,$$

$$\langle ms, mt, DL \rangle, \langle mt, ms, DL \rangle,$$

$$\langle ms, mt, c \rangle, \langle mt, ms, c \rangle, \langle ms, mt, p \rangle, \langle mt, ms, p \rangle$$

$$\langle ms, mt, c, wt \rangle, \langle mt, ms, c, wt \rangle, \langle ms, mt, c, ws \rangle, \langle mt, ms, c, ws \rangle,$$

$$\langle ms, mt, c, ws, wt \rangle, \langle ms, mt, c, wt, ws \rangle, \langle mt, ms, c, ws, wt \rangle, \langle mt, ms, c, wt, ws \rangle \}$$

The deadlock scenarios are seen in the traces where process S is waiting for process T and vice versa. This shows that no synchronisation event in set Y is communicated. The operator has been tested with many process scenarios. Each of the result sets of the test scenarios were tested for trace refinement using ProCSP. This was done by checking that the generated traces are trace refinements of the specification in CSP_M and then by confirming that the whole state space is covered, a graphical function provided by ProCSP.

It has been found that all the scenarios tested were trace refinements of the specification. The implementation is therefore accepted, based on the unit tests performed for each of the possible outcomes for the step law of generalised parallelism.

5.2.4.2 Optional parallel implementation

The generalised parallel function was modified to lift the synchronisation restrictions to allow for an optional parallel implementation. Pseudo code of the recursive optional parallel operator function is given in Algorithm 2. The function receives two strings, s and t , representing traces as input, a synchronisation set X and an initially empty history of traces. The start of the function deals with the exit conditions of the recursive function where a process is allowed to continue after the other process has finished. The recursion ends when both trace sets have finished producing traces. The rest of the function deals with the step laws of optional parallelism. At first, in lines 11 to 15, it is checked if $s[0]$ is contained in the synchronisation set X . If it is equal to $t[0]$, the first condition of the step law of Equation 3.32 is met. The second condition of the step law, where either process may proceed if their events are not in the synchronisation set X , is handled in lines 17 to 21. The last two conditions are handled in lines 22 to 24 and lines 25 to 27. These conditions are unique to optional parallelism where

a process may proceed, regardless if its event is contained in the synchronisation set X .

5.2.4.2.1 Validation

As with the generalised parallel operator, the optional parallel operator was first validated by using simple examples of two communicating processes running concurrently which tested the binary relationship of the operator. The same examples of the generalised parallel operator of equations 5.3 and 5.4 are used to illustrate the differences between the two operators.

The trace set scenario for each process contained a finite set of traces for each process to complete one cycle of execution. All the possible traces between processes P and Q in optional parallel, $traces(P \uparrow_X Q)$, for their finite sets of traces, were generated by the optional parallel function to be:

```
{ <>, <mp>, <mq>
  <mp, c>, <mp, mq>, <mq, mp>, <mq, c>
  <mp, c, wp>, <mp, c, mq>, <mp, mq, c>, <mq, mp, c>, <mq, c, mp>, <mq, c, wq>
  <mp, c, wp, mq>, <mp, c, mq, wp>, <mp, c, mq, c>, <mp, mq, c, wp>, <mp, mq, c, wq>,
  <mp, mq, c, c>, <mq, mp, c, wp>, <mq, mp, c, wq>, <mq, mp, c, c>, <mq, c, mp, c>,
  <mq, c, mp, wq>, <mq, c, wq, mp>,
  <mp, c, wp, mq, c>, <mp, c, mq, wp, c>, <mp, c, mq, c, wp>, <mp, c, mq, c, wq>,
  <mp, mq, c, wp, wq>, <mp, mq, c, wq, wp>, <mp, mq, c, wp, c>, <mp, mq, c, c, wp>,
  <mp, mq, c, c, wq>, <mp, mq, c, wq, c>, <mq, mp, c, wp, wq>, <mq, mp, c, wq, wp>,
  <mq, mp, c, wp, c>, <mq, mp, c, c, wp>, <mq, mp, c, c, wq>, <mq, mp, c, wq, c>,
  <mq, c, mp, c, wp>, <mq, c, mp, c, wq>, <mq, c, mp, wq, c>, <mq, c, wq, mp, c>
  <mp, c, wp, mq, c, wq>, <mp, c, mq, wp, c, wq>, <mp, c, mq, c, wp, wq>,
  <mp, c, mq, c, wq, wp>, <mp, mq, c, wp, c, wq>, <mp, mq, c, c, wp, wq>,
  <mp, mq, c, c, wq, wp>, <mp, mq, c, wq, c, wp>, <mq, mp, c, wp, c, wq>,
  <mq, mp, c, c, wp, wq>, <mq, mp, c, c, wq, wp>, <mq, mp, c, wq, c, wp>,
  <mq, c, mp, c, wp, wq>, <mq, c, mp, c, wq, wp>, <mq, c, mp, wq, c, wp>,
  <mq, c, wq, mp, c, wp> }
```

From the results, there are 61 trace sets. It can be seen that in some traces, only one $\{c\}$

Algorithm 2 Pseudo code for optional parallelism of [17].

```

1: procedure OPTPARALLEL(TraceSet s, TraceSet t, Set X, TraceSet traceHistory)
2:   if s.Count == 0 and t.Count == 0 then
3:     return
4:   else if s.Count == 0 then
5:     traceHistory.Append(t[0])
6:     OPTPARALLEL(s, t[1..end], X, traceHistory)
7:   else if t.Count == 0 then
8:     traceHistory.Append(s[0])
9:     OPTPARALLEL(s[1..end], t, X, traceHistory)
10:  else
11:    if s[0] and t[0] is contained in X then
12:      if s[0] == t[0] then
13:        traceHistory.Append(s[0])
14:        OPTPARALLEL(s[1..end], t[1..end], X, traceHistory)
15:      end if
16:    else
17:      if s[0] is contained in t.Alphabet then
18:        traceHistory.Append(s[0])
19:        OPTPARALLEL(s[1..end], t, X, traceHistory)
20:        traceHistory.Append(t[0])
21:        OPTPARALLEL(s, t[1..end], X, traceHistory)
22:      else if s[0] is not contained in t.Alphabet then
23:        traceHistory.Append(s[0])
24:        OPTPARALLEL(s[1..end], t, X, traceHistory)
25:      else if t[0] is not contained in s.Alphabet then
26:        traceHistory.Append(t[0])
27:        OPTPARALLEL(s, t[1..end], X, traceHistory)
28:      end if
29:    end if
30:  end if
31: end procedure

```

is communicated which indicates synchronisation, and in other cases, two $\{c\}$ events occur in the same trace set. This happens because optional parallelism allows two processes to synchronise when they can, and to proceed as if they are independent if they are not able to synchronise. The results presented for $traces(P \uparrow_X Q)$ represent the assumptions made in Chapter 3, where unclear properties of the *OptPar* definition of [17] were defined, based on the requirements of optional parallelism.

By applying *OptPar* in the example of Equation 5.4, the same traces are generated as if the processes were communicating independently. The trace results are left out because they contain every possible permutation of the combined traces of the processes S and T . There are no trace sequences leading to deadlock and is confirmed by the overall goal of optional parallelism, where processes are not required to synchronise in order to proceed with execution, but are allowed to synchronise when they are able to do so.

After black box testing, the output was inspected and compared against the calculations made by hand as there is no tool available yet which can be used as an external evaluation tool. To aid the software testing process, the following checks were done on multiple trace examples.

- Check that all traces are possible as per the step law of optional parallelism given in Equation 3.32. (This is a tedious task to perform by hand, but is the only way to check the trace validities.)
- Check that traces exist where synchronisation occurred, if theoretically possible.
- Check that interleaving of the synchronisation events are observed.
- Check that $traces(P \uparrow_X Q) = traces(P ||| Q)$ for $X = \{\}$.
- Check if the traces of general parallelism are equal to the traces of optional parallelism when the independent traces are removed [17]:

$$traces(P \parallel_X Q) \setminus traces(P ||| Q) = traces(P \uparrow_X Q) \setminus traces(P ||| Q).$$
- Check that no deadlock conditions are observed.

To conclude the black box testing process of the optional parallel implementation, the trace re-

finement relationship between optional parallelism and general parallelism defined in [17] was checked for each test example. The refinement relationship is shown in Equation 5.5.

$$\text{traces}(R_1 \uparrow_X R_2) \sqsubseteq_T \text{traces}(R_1 \parallel_X R_2) \quad (5.5)$$

Equation 5.5 states that the traces of optional parallelism are refined by the traces of generalised parallelism. This is so by definition, where the general parallel operator is more restricted than the optional parallel operator, requiring synchronising processes to do so jointly

5.2.5 Simulated trace generation

A simulator was developed to test the theory of WSN nodes communicating with each other using $OptPar^T$. A set of threads are spawned for each of the node- and channel processes given in a list as input. The Task Parallel Library ³ (TPL) [111] of the Microsoft .Net Framework was used to simulate each node- and channel process in its own thread. The TPL allows for a thread to be assigned to an individual CPU core which enables the simulated node processes to run completely independently on its own CPU core, without having to share processing resources with other threads. This is obviously limited to the number of CPU cores available as well as the number of dedicated threads per core. The concept is illustrated with examples of small node counts. It was later concluded that due to the non-critical simulated timing between the processes, it was not needed for the simulator to schedule each node- and channel process thread to its own CPU core because the interval of communication is set to be slower for human observation. The TPL will do its best to ensure that each thread gets sufficient CPU time. Each node process communicates to each of its connected node processes at random intervals, through the channels. The channels randomly elects to drop the communication or to allow it to pass through from sender to receiver, simulating a process as ready to engage or not. The probability of a decision made by the channel process can be defined as was discussed in §4.4.2. The synchronisation construct used for thread synchronisation between the node- and channel processes makes use of events which notifies listening threads that it should respond. If it does not respond, the thread blocks indefinitely.

The node process implementation is done with a *while* loop, executing a Finite State Machine (FSM) which performs the events as defined in the node process equations of Chapter

³<http://msdn.microsoft.com/concurrency>

6. The *while* loop is aborted when the process gets a STOP command from the simulator environment. The node process has an internal list of connected channel processes. Due to the disconnected definition of the optional parallel translation, the processes are only directly connected to channels, and the channels serve as the communication links between the processes. A channel is added for each element in the synchronisation set between processes. This means that there is one channel process between each of the connected processes. Synchronisation between the node processes and the channel processes are done with signals and threads waiting on these signals. This is done in C# with the *AutoResetEvent*. Although not the most economic thread synchronisation mechanism in terms of overhead, it provides a simple implementation. When a process is in its transmit state, the $\{communicate\}$ event is observed after a successful synchronisation with a particular channel. This is done after the process has requested a channel synchronisation and the channel responds by setting the *AutoResetEvent* on which the node process is blocking.

The channel process implementation has the same design as the node process implementation. The FSM of the channel is implemented according to the channel definition of Equation 4.8. When a synchronisation request is received from a connected node process, it gets handled when the channel is in a state to process synchronisation requests. The process requesting the synchronisation is in an indefinite blocked state until the event is set by the channel. After the synchronisation is completed with the process, indicating the $\{internal\}$ event of Equation 4.8, the channel randomly decides to forward the synchronisation to the process connected to its $\{external\}$ event. The frequency of the dropped packets are simulated using the pseudo random function of the Microsoft .Net Framework which is based on the subtractive random number generator algorithm from [112].

5.3 CHAPTER SUMMARY

This chapter was dedicated to the new software tools developed in aid of the optional parallel theoretical validation process. OptoCSP and OpTrace can be used as tools for further research of optional parallelism. Their implementation was done in such a manner that it can easily be updated and modified for specific use. The tools were tested as far as was needed to be used for the examples in this dissertation.

CHAPTER 6

OPTIONAL PARALLEL TEST SCENARIO DESCRIPTION

The applicability of $OptPar^T$ is to be tested with specific WSN system scenarios, which are driven by the different categories of WSN topologies or graph structures ¹. The scenarios will be used to test optional parallelism from two perspectives. The first is from a theoretical perspective, where the traces of the WSN systems will be compared between $OptPar$ and $OptPar^T$. The second perspective is from an practical perspective where the topologies will be converted to CSP definitions which can be used in existing model-checkers.

Trace refinement checks are done for several scenarios to see if a trace relation exists between the optional parallel operators $\overset{\sim}{\parallel}_X$ and $\overset{\uparrow}{\parallel}_X$ for the same network. With the absence of an implementation of $OptPar$ of [17], a general trace refinement proof between the two optional parallel operators becomes difficult. The topologies used in this chapter are selected from the most common WSN graph structures found in practise. More complex networks can be constructed from a combination of the basic topologies presented here.

6.1 TOPOLOGY SCENARIOS

WSNs can consist of hundreds of sensor nodes, and sometimes do not have a physical architecture as the nodes are often randomly deployed. WSN routing protocols create a logical topology between the sensor nodes of which four main groups exist, flat topologies, cluster topologies, chain topologies and tree topologies [113]. The logical topology defines the communication links between the nodes and it does not necessarily mean that nodes which are in

¹The terms *graph structure* and *topology* are used interchangeably in this chapter.

each others' range have an actual communication link between them. This is defined by the WSN routing protocols, a topic outside of the scope of this section. The scenarios presented in this section emulates a converged topology for a specific network, i.e. the topologies are assumed to be fixed for the analysis.

The WSN topologies of the scenarios are presented with binary connections between the sensor nodes. The interpretation of these binary connections will be explained for each scenario and the CSP system definitions will be given.

The topology scenarios are by no means chosen to point out advantages or disadvantages of the topology groups, but merely to illustrate the applicability of optional parallelism to model it in CSP.

Where applicable, the scenarios will be detailed with directional communication and hence directional synchronisation, as discussed in §4.3. Three different sub-scenarios will be defined, where applicable, using directional synchronisation:

- Unidirectional, i.e. broadcasting. Transmitter node with multiple independent receiver nodes.
- Bidirectional, half-duplex synchronisation. Normal bidirectional synchronisation events between two processes.
- Individual unidirectional in both directions, simplex synchronisation. Two independent simplex synchronisation events between two processes.

Full-duplex communication will be left out as the models have these same results as the half-duplex models in terms of system structure, trace refinement and deadlock freedom, but with added complexity and execution times on model-checking with ProCSP. Note that the channel definition for half-duplex communication in Equation 4.28 is structurally the same as the one for full-duplex in Equation 4.29, with the only difference being the choice of operator between the sending and receiving atomic channel artefacts. External choice (\square) is used for half-duplex synchronisation and interleaving (\parallel) is used for full-duplex synchronisation. Hence, to keep the test scenarios as short and tidy as possible, full-duplex communication has been left out of the analysis.

The general behaviour of the processes used to model the WSN nodes are chosen to have basic functionality. WSN nodes are modelled as processes which measures or senses an environment variable (*measure*), communicates it with its neighbours (*comm*), and lastly performs some internal operations (*work*). The nodes are also modelled to be able to sleep indefinitely (*sleep*) to simulate a node which is not able to engage in synchronised communication events. In some scenarios, some of the non-synchronised CSP events are left out purely to simplify the models and give meaning to the processes and their applicability to their chosen WSN graph structures.

6.1.1 Flat topology

6.1.1.1 Point-to-point

The point-to-point connection, depicted in Figure 6.1 for processes P and Q , is the most basic connection. It is added as a control to see if the most basic connection pass the trace refinement checks of the two optional parallel operators as well as the conversion from optional parallelism to classical CSP operators. Broadcasting in this scenario is the same as a single simplex synchronisation event, either from P to Q or from Q to P . A bidirectional communication link between processes P and Q can also exist. As mentioned, the bidirectional synchronisation can either be implemented with 2 individual simplex synchronisation events or a single half-duplex synchronisation event. This is detailed in the sub-scenarios that follow.

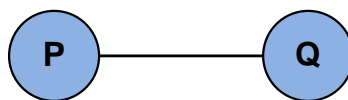


Figure 6.1: Point-to-point connection between processes P and Q .

6.1.1.1.1 Broadcasting

Figure 6.2 shows broadcasting between processes P and Q and its adjacency list is given in Listing 6.1.

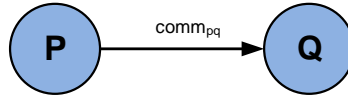


Figure 6.2: Broadcasting between processes P and Q .

(P, Q)

Listing 6.1: Adjacency list for broadcasting from node P to Q .

The CSP process definitions for Figure 6.2 are given in Equation 6.1.

$$\begin{aligned}
 P &:= (\text{measure}_p \rightarrow \overrightarrow{\text{comm}_{pq}} \rightarrow \text{work}_p \rightarrow P) \sqcap (\text{sleep}_p \rightarrow P) \\
 Q &:= (\text{measure}_q \rightarrow \overleftarrow{\text{comm}_{pq}} \rightarrow \text{work}_q \rightarrow Q) \sqcap (\text{sleep}_q \rightarrow Q)
 \end{aligned} \tag{6.1}$$

The network definition is given in Equation 6.2.

$$WSN := P \quad \overset{\sim}{\parallel} \quad Q \tag{6.2}$$

$\{ \text{comm}_{pq} \}$

The synchronisation set of Equation 6.2 shows that process P and Q optionally synchronises on the comm_{pq} event. This means that a channel artefact should be added to each process definition for P and Q for this synchronisation event. The arrow on top of the synchronisation event indicates the channel artefact to be used, either broadcasting, simplex, half-duplex or full-duplex (see §4.3). In this example, the broadcasting and simplex solutions are the same as per Table 4.1 and a C_{transmit} channel artefact of Equation 4.23 will be linked to node P with a C_{receive} channel artefact of Equation 4.24 linked to node Q . The channel artefacts added to processes P and Q will be called CHP and CHQ respectively. The comm_{pq} event of process P will be renamed to a common event between process P and its channel CHP to form P' . This event will be renamed to comm_{CHP} and corresponds to the *internal* event of Equation 4.23. The *external* event of C_{transmit} of Equation 4.23 corresponds to the synchronisation

event $comm_{pq}$ and will be renamed to it. The new definition of process P , called P' , as well as its channel artefact definition are given below in Equation 6.3.

$$\begin{aligned}
 P' &:= (measure_p \rightarrow comm_{CHP} \rightarrow work_p \rightarrow P') \sqcap (sleep_p \rightarrow P') \\
 CHP &:= (comm_{CHP} \rightarrow ((comm_{pq} \rightarrow CHP) \sqcap CHP))
 \end{aligned} \tag{6.3}$$

The same is done for process Q to form Q' and CHQ as shown in Equation 6.4 below.

$$\begin{aligned}
 Q' &:= (measure_q \rightarrow comm_{CHQ} \rightarrow work_q \rightarrow Q') \sqcap (sleep_q \rightarrow Q') \\
 CHQ &:= comm_{pq} \rightarrow ((comm_{CHQ} \rightarrow CHQ) \sqcap CHQ)
 \end{aligned} \tag{6.4}$$

From Equations 6.3 and 6.4, it can be seen that P' and Q' do not share an event any more and are thus independent (interleaved). The channels CHP and CHQ share the $comm_{pq}$ event, which is the synchronisation event of the initial system definition of Equation 6.2. From here, it is easy to write the translated system in the form given in Equation 4.21. This is given below:

$$P \underset{\{comm_{pq}\}}{\parallel} Q = (P' \parallel Q') \underset{\{comm_{CHP}, comm_{CHQ}\}}{\parallel} \left(CHP \underset{comm_{pq}}{\parallel} CHQ \right) \tag{6.5}$$

These calculations are omitted for the rest of the scenario examples of this chapter as Op-toCSP can be used to derive the translations.

6.1.1.1.2 Bidirectional - Half-duplex

Figure 6.3 depicts a half-duplex synchronisation event between processes P and Q . This means that any of the two processes can initiate a synchronisation. The adjacency list is given in Listing 6.2.



Figure 6.3: Bidirectional synchronisation between processes P and Q using a half-duplex synchronisation event.

{P, Q}

Listing 6.2: Adjacency list for half-duplex synchronisation between processes P and Q .

The CSP process definitions are given in Equation 6.6 and the network definition for optional parallelism in Equation 6.7.

$$\begin{aligned}
 P &:= (measure_p \rightarrow \overleftarrow{comm} \rightarrow work_p \rightarrow P) \sqcap (sleep_p \rightarrow P) \\
 Q &:= (measure_q \rightarrow \overleftarrow{comm} \rightarrow work_q \rightarrow Q) \sqcap (sleep_q \rightarrow Q)
 \end{aligned}
 \tag{6.6}$$

$$WSN := P \parallel_{\{comm\}} Q
 \tag{6.7}$$

This allows P to have a $comm$ event even if Q is never ready to synchronise on $comm$ and vice versa.

6.1.1.1.3 Bidirectional - Simplex

Figure 6.4 depicts bidirectional synchronisation between processes P and Q using 2 simplex synchronisation events. This means that any of the two processes can initiate a synchronisation. The adjacency list is given in Listing 6.3.

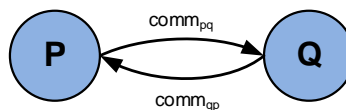


Figure 6.4: Bidirectional synchronisation between processes P and Q using 2 simplex synchronisation events.

(P, Q)
 (Q, P)

Listing 6.3: Adjacency list for bidirectional synchronisation between processes P and Q using 2 simplex synchronisation events.

The CSP process definitions are given in Equation 6.8 and the network definition for optional parallelism in Equation 6.9.

$$\begin{aligned}
 P &:= (\text{measure}_p \rightarrow \overrightarrow{\text{comm}}_{pq} \rightarrow \text{work}_p \rightarrow P) \sqcap (\text{measure}_p \rightarrow \overleftarrow{\text{comm}}_{qp} \rightarrow \text{work}_p \rightarrow P) \\
 &\quad \sqcap (\text{sleep}_p \rightarrow P) \\
 Q &:= (\text{measure}_q \rightarrow \overleftarrow{\text{comm}}_{pq} \rightarrow \text{work}_q \rightarrow Q) \sqcap (\text{measure}_q \rightarrow \overrightarrow{\text{comm}}_{qp} \rightarrow \text{work}_q \rightarrow Q) \\
 &\quad \sqcap (\text{sleep}_q \rightarrow Q)
 \end{aligned} \tag{6.8}$$

$$WSN := P \quad \overset{\sim}{\parallel} \quad Q \\
 \{ \text{comm}_{pq}, \text{comm}_{qp} \} \tag{6.9}$$

6.1.1.2 Fully-connected mesh topology

A mesh topology allows sensor nodes in a WSN to not only send their own data to the sink, but also to relay the data of its neighbours to the sink. This allows for route redundancy in the case of a node failure.

The advantages of optional parallelism is most visible in fully connected mesh networks. When a node process broadcasts a message, all available processes will synchronise with the transmitting process. Any of the connected processes which are not ready to synchronise, are allowed to opt out of synchronisation and the rest of the synchronising processes can continue with their execution. When modelled explicitly with the help of different event names in classical CSP, it comes at the cost of many extra states because all permutations of possible synchronisations need to be considered.

A 4-node fully-connected mesh topology is given in Figure 6.5. This topology can be obtained from two approaches. The first approach is as per the definition of a mesh topology, all the nodes have a point-to-point connection with each of the other nodes in the network. From a practical perspective, the communication between the nodes is defined to be a set of point-to-point connections which makes the network more redundant because messages can be relayed via another node if a communication link fails. The second approach comes from a modelling perspective. The translation of an n-ary relation to a set of binary relations, as was mentioned in §4.2.2, breaks a hyper-edge up into multiple point-to-point connections. In this approach, the nodes seem as if they are connected in point-to-point connections, but

each node is *listening* for communication from any other node. In CSP, this is achieved by enabling all nodes to synchronise on a network transmission, regardless where it is from. Stated differently, when a process sends out a message, it is broadcast to all the other nodes at the same time, as opposed to unicast where a message is sent to each neighbouring node independently.

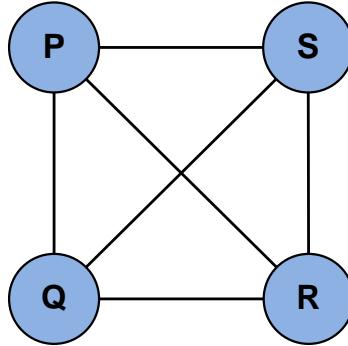


Figure 6.5: Fully connected processes P , Q , R and S .

The 3 sub-scenarios detailing broadcasting, half-duplex bidirectional and simplex bidirectional synchronisation are detailed below.

6.1.1.2.1 Broadcasting

Figure 6.6 shows the different broadcasting approaches between processes P , Q , R and S . Each process has a unique broadcasting event to the other processes. A broadcaster is identified and the other processes are only able to listen and synchronise if the broadcaster has a broadcast event. The nodes are modelled so that they can send their measurements to their neighbours, or receive a measurement from a neighbour and perform internal operations. In the case where a measurement is received, no attempt to transmit a measurement will be made until a new measurement has been sensed. It is shown in Figure 6.6 that a directional hyper-edge exists for the broadcasting event from the particular broadcasting node. A common broadcasting event is shared between the broadcaster and the sensor nodes, but not between the sensor nodes themselves. The adjacency list is given in Listing 6.4.

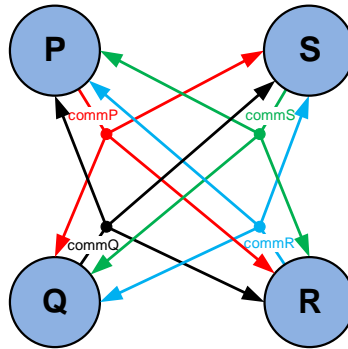


Figure 6.6: Broadcasting approaches between processes P , Q , R and S .

(P, Q, R, S)
 (Q, P, R, S)
 (R, P, Q, S)
 (S, P, Q, R)

Listing 6.4: Adjacency list for broadcasting synchronisation between processes P , Q , R and S .

The CSP process definitions for Figure 6.6 are given in Equation 6.10.

$$\begin{aligned}
 P &:= ((\overrightarrow{measure_p} \rightarrow \overrightarrow{commP} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{commQ} \rightarrow work_p \rightarrow P)) \\
 &\quad \sqcap (\overleftarrow{commR} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{commS} \rightarrow work_p \rightarrow P)) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((\overrightarrow{measure_q} \rightarrow \overrightarrow{commQ} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{commP} \rightarrow work_q \rightarrow Q)) \\
 &\quad \sqcap (\overleftarrow{commR} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{commS} \rightarrow work_q \rightarrow Q)) \\
 &\quad \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((\overrightarrow{measure_r} \rightarrow \overrightarrow{commR} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{commP} \rightarrow work_r \rightarrow R)) \\
 &\quad \sqcap (\overleftarrow{commQ} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{commS} \rightarrow work_r \rightarrow R)) \\
 &\quad \sqcap (sleep_r \rightarrow R) \\
 S &:= ((\overrightarrow{measure_s} \rightarrow \overrightarrow{commS} \rightarrow work_s \rightarrow S) \sqcap (\overleftarrow{commP} \rightarrow work_s \rightarrow S)) \\
 &\quad \sqcap (\overleftarrow{commQ} \rightarrow work_s \rightarrow S) \sqcap (\overleftarrow{commR} \rightarrow work_s \rightarrow S)) \\
 &\quad \sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.10}$$

The network definition is given in Equation 6.11.

$$\begin{aligned}
 WSN &:= P \parallel_X Q \parallel_X R \parallel_X S, \text{ where} \\
 X &:= \{commP, commQ, commR, commS\}
 \end{aligned} \tag{6.11}$$

6.1.1.2.2 Bidirectional - Half-duplex

Figure 6.7 depicts the communication links between processes P , Q , R and S as a hypergraph. This means that any of the processes can initiate a synchronisation and that it can happen in any direction. The processes are modelled to all measure an environment variable and exchange their data with each other simultaneously in a synchronised manner. The adjacency list is given in Listing 6.5.

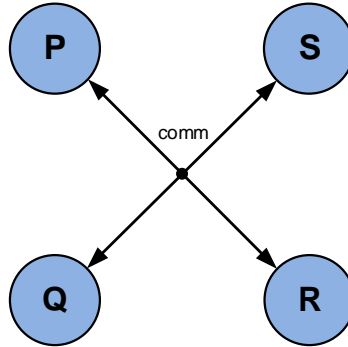


Figure 6.7: Bidirectional synchronisation between processes P , Q , R and S using half-duplex synchronisation.

$\{P, Q, R, S\}$

Listing 6.5: Adjacency list for half-duplex synchronisation between processes P , Q , R and S .

The CSP process definitions are given in Equation 6.12 and the network definition for optional parallelism in Equation 6.13.

$$\begin{aligned}
 P &:= (measure_p \rightarrow \overleftarrow{comm} \rightarrow work_p \rightarrow P) \sqcap (sleep_p \rightarrow P) \\
 Q &:= (measure_q \rightarrow \overleftarrow{comm} \rightarrow work_q \rightarrow Q) \sqcap (sleep_q \rightarrow Q) \\
 R &:= (measure_r \rightarrow \overleftarrow{comm} \rightarrow work_r \rightarrow R) \sqcap (sleep_r \rightarrow R) \\
 S &:= (measure_s \rightarrow \overleftarrow{comm} \rightarrow work_s \rightarrow S) \sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.12}$$

$$WSN := P \underset{\{comm\}}{\parallel} Q \underset{\{comm\}}{\parallel} R \underset{\{comm\}}{\parallel} S \tag{6.13}$$

This allows P to have a $comm$ event even if Q , R and S are never ready to synchronise on $comm$ and vice versa.

6.1.1.2.3 Bidirectional - Simplex

Figure 6.8 depicts bidirectional synchronisation between processes P , Q , R and S using 2 simplex synchronisation events between all of the processes. The n -ary hyper-graph is converted into a normal graph with binary connections as was previously mentioned. This means that there exists a unique channel between any two processes with a separate channel for each of the possible directions between the two connected processes. The adjacency list for this scenario is given in Listing 6.6.

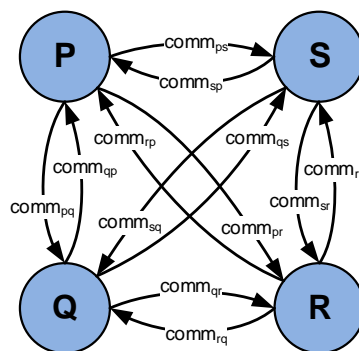


Figure 6.8: Bidirectional synchronisation between processes P , Q , R and S using 2 simplex synchronisation events.

```
(P, Q)
(P, R)
(P, S)
(Q, P)
(Q, R)
(Q, S)
(R, P)
(R, Q)
(R, S)
(S, P)
(S, Q)
(S, R)
```

Listing 6.6: Adjacency list for bidirectional synchronisation between processes P , Q , R and S using 2 simplex synchronisation events.

The CSP process definitions are given in Equation 6.14 and the network definition for optional parallelism in Equation 6.15.

$$\begin{aligned}
 P &:= (\text{measure}_p \rightarrow \overrightarrow{\text{comm}}_{pq} \rightarrow \text{work}_p \rightarrow P) \sqcap (\text{measure}_p \rightarrow \overleftarrow{\text{comm}}_{qp} \rightarrow \text{work}_p \rightarrow P) \\
 &\sqcap (\text{measure}_p \rightarrow \overrightarrow{\text{comm}}_{pr} \rightarrow \text{work}_p \rightarrow P) \sqcap (\text{measure}_p \rightarrow \overleftarrow{\text{comm}}_{rp} \rightarrow \text{work}_p \rightarrow P) \\
 &\sqcap (\text{measure}_p \rightarrow \overrightarrow{\text{comm}}_{ps} \rightarrow \text{work}_p \rightarrow P) \sqcap (\text{measure}_p \rightarrow \overleftarrow{\text{comm}}_{sp} \rightarrow \text{work}_p \rightarrow P) \\
 &\sqcap (\text{sleep}_p \rightarrow P) \\
 Q &:= (\text{measure}_q \rightarrow \overrightarrow{\text{comm}}_{qp} \rightarrow \text{work}_q \rightarrow Q) \sqcap (\text{measure}_q \rightarrow \overleftarrow{\text{comm}}_{pq} \rightarrow \text{work}_q \rightarrow Q) \\
 &\sqcap (\text{measure}_p \rightarrow \overrightarrow{\text{comm}}_{qr} \rightarrow \text{work}_p \rightarrow P) \sqcap (\text{measure}_p \rightarrow \overleftarrow{\text{comm}}_{rq} \rightarrow \text{work}_q \rightarrow Q) \\
 &\sqcap (\text{measure}_p \rightarrow \overrightarrow{\text{comm}}_{qs} \rightarrow \text{work}_p \rightarrow P) \sqcap (\text{measure}_p \rightarrow \overleftarrow{\text{comm}}_{sq} \rightarrow \text{work}_q \rightarrow Q) \\
 &\sqcap (\text{sleep}_q \rightarrow Q) \\
 R &:= (\text{measure}_r \rightarrow \overrightarrow{\text{comm}}_{rp} \rightarrow \text{work}_r \rightarrow R) \sqcap (\text{measure}_r \rightarrow \overleftarrow{\text{comm}}_{pr} \rightarrow \text{work}_r \rightarrow R) \\
 &\sqcap (\text{measure}_r \rightarrow \overrightarrow{\text{comm}}_{rq} \rightarrow \text{work}_r \rightarrow R) \sqcap (\text{measure}_r \rightarrow \overleftarrow{\text{comm}}_{qr} \rightarrow \text{work}_r \rightarrow R) \\
 &\sqcap (\text{measure}_r \rightarrow \overrightarrow{\text{comm}}_{rs} \rightarrow \text{work}_r \rightarrow R) \sqcap (\text{measure}_r \rightarrow \overleftarrow{\text{comm}}_{sr} \rightarrow \text{work}_r \rightarrow R) \\
 &\sqcap (\text{sleep}_r \rightarrow R) \\
 S &:= (\text{measure}_s \rightarrow \overrightarrow{\text{comm}}_{sp} \rightarrow \text{work}_s \rightarrow S) \sqcap (\text{measure}_s \rightarrow \overleftarrow{\text{comm}}_{ps} \rightarrow \text{work}_s \rightarrow S) \\
 &\sqcap (\text{measure}_s \rightarrow \overrightarrow{\text{comm}}_{sq} \rightarrow \text{work}_s \rightarrow S) \sqcap (\text{measure}_s \rightarrow \overleftarrow{\text{comm}}_{qs} \rightarrow \text{work}_s \rightarrow S) \\
 &\sqcap (\text{measure}_s \rightarrow \overrightarrow{\text{comm}}_{sr} \rightarrow \text{work}_s \rightarrow S) \sqcap (\text{measure}_s \rightarrow \overleftarrow{\text{comm}}_{rs} \rightarrow \text{work}_s \rightarrow S) \\
 &\sqcap (\text{sleep}_s \rightarrow S)
 \end{aligned} \tag{6.14}$$

$$WSN := \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{\text{comm}_{pq}, \text{comm}_{pr}, \text{comm}_{ps}, \text{comm}_{qp}, \text{comm}_{rp}, \text{comm}_{sp}, \text{work}_p, \text{measure}_p, \text{sleep}_p\})$$

$$(N_2, \alpha N_2) = (Q, \{\text{comm}_{qp}, \text{comm}_{qr}, \text{comm}_{qs}, \text{comm}_{pq}, \text{comm}_{rq}, \text{comm}_{sq}, \text{work}_q, \text{measure}_q, \text{sleep}_q\})$$

$$(N_3, \alpha N_3) = (R, \{\text{comm}_{rp}, \text{comm}_{rq}, \text{comm}_{rs}, \text{comm}_{pr}, \text{comm}_{qr}, \text{comm}_{qs}, \text{work}_r, \text{measure}_r, \text{sleep}_r\})$$

$$(N_4, \alpha N_4) = (S, \{\text{comm}_{sp}, \text{comm}_{sq}, \text{comm}_{sr}, \text{comm}_{ps}, \text{comm}_{qs}, \text{comm}_{rs}, \text{work}_s, \text{measure}_s, \text{sleep}_s\})$$

(6.15)

6.1.2 Cluster

A cluster topology can be arranged to form a star topology if there is one cluster head (or sink node) and several directly connected sensor nodes, i.e. the node hierarchy is only one level deep, as shown in Figure 6.9. In this scenario, the cluster head is allowed to broadcast a message to all its connected sensor nodes, but if a sensor node wishes to respond to the cluster head, that communication is point-to-point. Note that this structure differs from the example in §4.3 where the neighbour nodes are not defined to communicate back to the cluster head. With the communication strategy of this section, there exists a common synchronisation event, but it has a direction from the cluster head to the sensor nodes. The sensor nodes each have a direct, directional communication link back to the cluster head. This differs from the broadcasting scenarios depicted in Figure 6.6 of §6.1.1.2 where the receiving nodes are not defined to respond to the broadcaster. This approach is followed to limit the possible duplication of scenarios. Details regarding directional communication using CSP are given in §4.3, where the data direction is specifically modelled by the CSP definitions of the channel artefacts of optional parallelism.

In the cluster scenarios, the cluster heads will have different process structures than the sensor nodes. The process definitions of all the sensor nodes have the same structure, in contrast with the cluster head, which adapts to the converged logical topology where it has a separate communication event for responses from each logically connected sensor node.

The star topology is a common WSN topology where all nodes communicate with a sink node. The use of optional parallelism to model this topology is useful where the sink needs to communicate with all its nodes and only a subset of them can synchronise at that instance. This will allow only the processes which are ready to respond to the broadcast communication from the sink, to jointly synchronise on the broadcast event.

This topology tests the broadcasting communication approach where the sink node has many receive events, one for each connected process, but only one transmit broadcast event. The nodes on the edges are only aware of their point-to-point connections with the sink, and therefore has only one transmit and one receive event. The difference between the point-to-point approach will only be seen in the sink node process' definition where there will be a separate transmit event for each of its connected node processes. If the sink fails, the whole

network will be disconnected.

A 4-node star-topology is given in Figure 6.9. It is a simple scenario with one cluster head and 3 sensor nodes. In this scenario, the cluster head sends some information to the sensor nodes and receives measurements back from the sensor nodes.

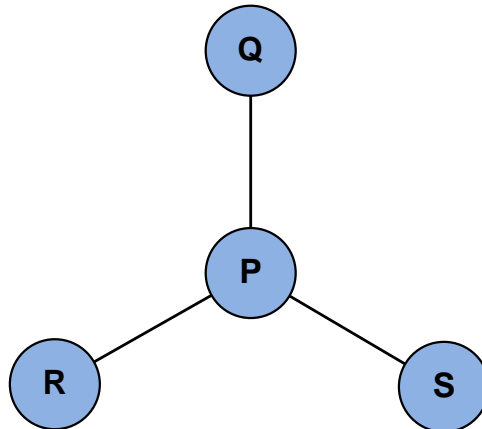


Figure 6.9: Star topology with processes P , Q , R and S .

The 3 sub-scenarios detailing broadcasting, half-duplex bidirectional and simplex bidirectional synchronisation are detailed below.

6.1.2.1 Broadcasting

In this cluster scenario, there is only one broadcasting process P . All the child nodes can only receive on the broadcasting event of process P . When the cluster of Figure 6.9 is redrawn showing its communication links as in Figure 6.10, it appears that a directional hyper-edge exists for the broadcasting event from the head node, P , but individual events exist for each of the possible responses from the sensor nodes (Q , R and S) back to node P . This is because the sensor nodes have no information about each other, but only with the cluster head P . A common broadcasting event is shared between the cluster head and the sensor nodes, but not between the sensor nodes themselves. This is why there are unique directional events from each of the sensor nodes back to the cluster head P . The adjacency list for this scenario is given in Listing 6.7.

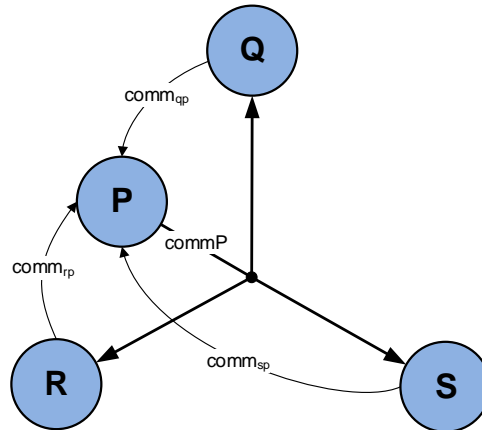


Figure 6.10: Directional communication links for the star topology of Figure 6.9.

(P, Q, R, S)
 (Q, P)
 (R, P)
 (S, P)

Listing 6.7: Adjacency list with broadcaster process P .

The CSP process definitions are given in Equation 6.16 and the network definition for optional parallelism in Equation 6.17.

$$\begin{aligned}
 P &:= ((\overrightarrow{commP} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{comm_{qp}} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{comm_{rp}} \rightarrow work_p \rightarrow P) \\
 &\quad \sqcap (\overleftarrow{comm_{sp}} \rightarrow work_p \rightarrow P)) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((\overleftarrow{commP} \rightarrow work_q \rightarrow Q) \sqcap (measure_q \rightarrow \overrightarrow{comm_{qp}} \rightarrow Q)) \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((\overleftarrow{commP} \rightarrow work_r \rightarrow R) \sqcap (measure_r \rightarrow \overrightarrow{comm_{rp}} \rightarrow R)) \sqcap (sleep_r \rightarrow R) \\
 S &:= ((\overleftarrow{commP} \rightarrow work_s \rightarrow S) \sqcap (measure_s \rightarrow \overrightarrow{comm_{sp}} \rightarrow S)) \sqcap (sleep_s \rightarrow S) \quad (6.16)
 \end{aligned}$$

$$WSN := \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{commP, comm_{qp}, comm_{rp}, comm_{sp}, work_p, sleep_p\})$$

$$(N_2, \alpha N_2) = (Q, \{commP, comm_{qp}, work_q, measure_q, sleep_q\})$$

$$(N_3, \alpha N_3) = (R, \{commP, comm_{rp}, work_r, measure_r, sleep_r\})$$

$$(N_4, \alpha N_4) = (S, \{commP, comm_{sp}, work_s, measure_s, sleep_s\}) \quad (6.17)$$

6.1.2.2 Bidirectional - Half-duplex

This scenario cannot be defined with a bidirectional hypergraph because there is only one broadcasting process defined. The closest this scenario can get to half-duplex bidirectional synchronisation is to define point-to-point connections between the cluster head and the sensor node processes. Figure 6.11 depicts the communication links between processes P , Q , R and S . These connections are all point-to-point, but synchronisation is done with optional parallelism. This scenario has the same structure when the general parallel operator is used. The adjacency list is given in Listing 6.8.

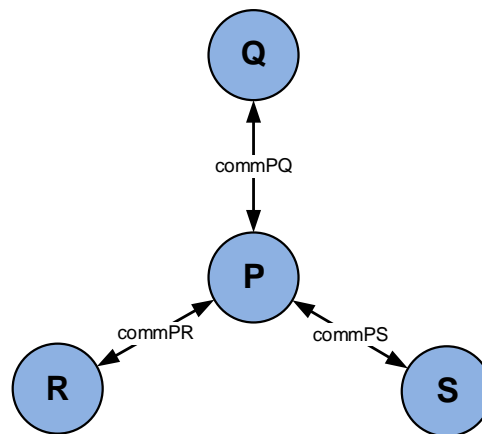


Figure 6.11: Bidirectional synchronisation between processes P , Q , R and S using half-duplex synchronisation.

{P,Q}
 {P,R}
 {P,S}

Listing 6.8: Adjacency list for half-duplex synchronisation between processes P , Q , R and S .

The CSP process definitions are given in Equation 6.18 and the network definition for optional parallelism in Equation 6.19.

$$\begin{aligned}
 P &:= ((measure_p \rightarrow \overleftarrow{commPQ} \rightarrow work_p \rightarrow P) \sqcap (measure_p \rightarrow \overleftarrow{commPR} \rightarrow work_p \rightarrow P) \\
 &\quad \sqcap (measure_p \rightarrow \overleftarrow{commPS} \rightarrow work_p \rightarrow P)) \sqcap (sleep_p \rightarrow P) \\
 Q &:= (measure_q \rightarrow \overleftarrow{commPQ} \rightarrow work_q \rightarrow Q) \sqcap (sleep_q \rightarrow Q) \\
 R &:= (measure_r \rightarrow \overleftarrow{commPR} \rightarrow work_r \rightarrow R) \sqcap (sleep_r \rightarrow R) \\
 S &:= (measure_s \rightarrow \overleftarrow{commPS} \rightarrow work_s \rightarrow S) \sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.18}$$

$$WSN := \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where}$$

$$\begin{aligned}
 (N_1, \alpha N_1) &= (P, \{commPQ, commPR, commPS, work_p, measure_p, sleep_p\}) \\
 (N_2, \alpha N_2) &= (Q, \{commPQ, work_q, measure_q, sleep_q\}) \\
 (N_3, \alpha N_3) &= (R, \{commPR, work_r, measure_r, sleep_r\}) \\
 (N_4, \alpha N_4) &= (S, \{commPS, work_s, measure_s, sleep_s\})
 \end{aligned} \tag{6.19}$$

6.1.2.3 Bidirectional - Simplex

Figure 6.12 depicts bidirectional synchronisation between processes P and each of the sensor nodes, Q , R and S , using 2 simplex synchronisation events between them. This means that there exists a separate channel for each of the possible directions between the two connected processes. The adjacency list for this scenario is given in Listing 6.9.

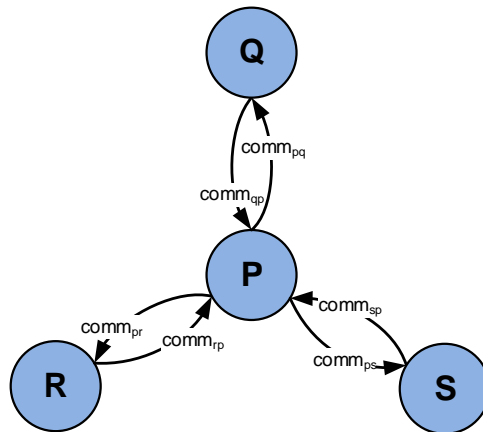


Figure 6.12: Bidirectional synchronisation between processes P , Q , R and S using 2 simplex synchronisation events.

(P, Q)
(P, R)
(P, S)
(Q, P)
(R, P)
(S, P)

Listing 6.9: Adjacency list for bidirectional synchronisation between processes P , Q , R and S using 2 simplex synchronisation events.

The CSP process definitions are given in Equation 6.20 and the network definition for optional parallelism in Equation 6.21.

$$\begin{aligned}
 P &:= (\overrightarrow{comm_{pq}} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{comm_{qp}} \rightarrow work_p \rightarrow P) \\
 &\sqcap (\overrightarrow{comm_{pr}} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{comm_{rp}} \rightarrow work_p \rightarrow P) \\
 &\sqcap (\overrightarrow{comm_{ps}} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{comm_{sp}} \rightarrow work_p \rightarrow P) \\
 &\sqcap (sleep_p \rightarrow P) \\
 Q &:= (measure_q \rightarrow \overrightarrow{comm_{qp}} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{comm_{pq}} \rightarrow work_q \rightarrow Q) \\
 &\sqcap (sleep_q \rightarrow Q) \\
 R &:= (measure_r \rightarrow \overrightarrow{comm_{rp}} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{comm_{pr}} \rightarrow work_r \rightarrow R) \\
 &\sqcap (sleep_r \rightarrow R) \\
 S &:= (measure_s \rightarrow \overrightarrow{comm_{sp}} \rightarrow work_s \rightarrow S) \sqcap (\overleftarrow{comm_{ps}} \rightarrow work_s \rightarrow S) \\
 &\sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.20}$$

$$WSN := \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{comm_{pq}, comm_{pr}, comm_{ps}, comm_{qp}, comm_{rp}, comm_{sp}, work_p, sleep_p\})$$

$$(N_2, \alpha N_2) = (Q, \{comm_{qp}, comm_{pq}, work_q, measure_q, sleep_q\})$$

$$(N_3, \alpha N_3) = (R, \{comm_{rp}, comm_{pr}, work_r, measure_r, sleep_r\})$$

$$(N_4, \alpha N_4) = (S, \{comm_{sp}, comm_{ps}, work_s, measure_s, sleep_s\}) \tag{6.21}$$

6.1.3 Chain

A chain topology involves the sensor nodes to be logically configured to form one or more transmission chains to the selected sink node. The data is aggregated from node to node until it is received by the sink node. The main objective of a chain topology is to reduce the energy used for data transmission. The chain topology for WSNs is typically used for border surveillance, transport route monitoring and pipeline monitoring. Although this approach is prone to broken links when a node fails, there are redundancy mechanisms that can be used to overcome this. These mechanisms fall outside of the scope of this scenario.

In a ring topology, each sensor node has two neighbours and communication is usually only in one direction. The examples here allow for communication in any direction at any time to keep the analysis as generic as possible. In these examples, when a node fails, or stops synchronising, the network effectively transforms into a chain topology. The ring topology scenarios are added as a theoretical test case from a modelling perspective where all the topologies are tested with optional parallelism in CSP. It also adds scenarios where possible circular wait conditions can occur, a condition that results in possible deadlock.

6.1.3.1 3-Node ring-topology

A 3-node ring-topology is given in Figure 6.13. This could be tested in both the point-to-point approach as well as the broadcasting communication approach. The point-to-point approach breaks the system into 3 distinct point-to-point connections between the node processes, resulting in a unique synchronisation set for each edge. It is up to the process definitions to allow the communication to occur in a specific direction.

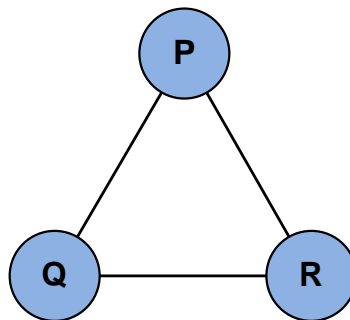


Figure 6.13: Ring topology with processes P , Q and R .

The 3 sub-scenarios detailing broadcasting, half-duplex bidirectional and simplex unidirectional synchronisation are detailed below.

6.1.3.1.1 Broadcasting

Figure 6.14 shows the different broadcasting approaches between processes P , Q and R . Each process has a unique broadcasting event to the other processes. A broadcaster is identified and the other processes are only able to listen and synchronise if the broadcaster

has a broadcast event. It is shown in Figure 6.14 that a directional hyper-edge exists for the broadcasting event from the particular broadcasting node. A common broadcasting event is shared between the broadcaster and the sensor nodes, but not between the sensor nodes themselves. The adjacency list is given in Listing 6.10. The 3-node ring scenario has the same broadcasting structure as the 4-node fully connected mesh topology discussed earlier in §6.1.1.2.

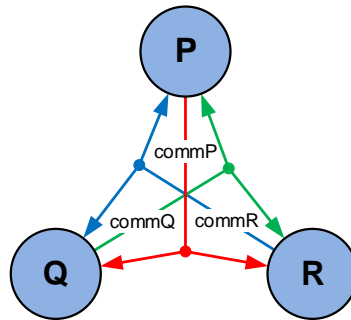


Figure 6.14: Ring topology with processes P , Q and R .

(P, Q, R) (Q, P, R) (R, P, Q)

Listing 6.10: Adjacency list for broadcasting synchronisation between processes P , Q and R .

The CSP process definitions for Figure 6.14 are given in Equation 6.22.

$$\begin{aligned}
 P &:= ((measure_p \rightarrow \overrightarrow{commP} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{commQ} \rightarrow work_p \rightarrow P) \\
 &\quad \sqcap (\overleftarrow{commR} \rightarrow work_p \rightarrow P)) \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((measure_q \rightarrow \overrightarrow{commQ} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{commP} \rightarrow work_q \rightarrow Q) \\
 &\quad \sqcap (\overleftarrow{commR} \rightarrow work_q \rightarrow Q)) \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((measure_r \rightarrow \overrightarrow{commR} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{commP} \rightarrow work_r \rightarrow R) \\
 &\quad \sqcap (\overleftarrow{commQ} \rightarrow work_r \rightarrow R)) \sqcap (sleep_r \rightarrow R)
 \end{aligned} \tag{6.22}$$

The network definition is given in Equation 6.23.

$$WSN := P \underset{X}{\parallel} Q \underset{X}{\parallel} R$$

$$X := \{commP, commQ, commR\} \quad (6.23)$$

6.1.3.1.2 Bidirectional - Half-duplex

Figure 6.15 depicts the communication links between processes P , Q and R . These connections are all point-to-point, but synchronisation is done with optional parallelism. This scenario has the same structure when the general parallel operator is used. The adjacency list is given in Listing 6.11.

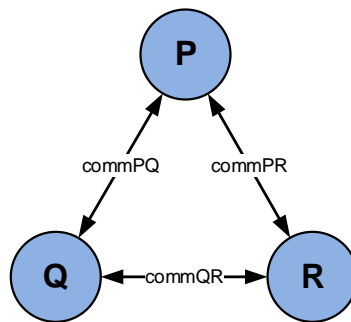


Figure 6.15: Bidirectional synchronisation between processes P , Q and R using half-duplex synchronisation.

```
{P,Q}
{P,R}
{Q,R}
```

Listing 6.11: Adjacency list for half-duplex synchronisation between processes P , Q and R .

The CSP process definitions are given in Equation 6.24 and the network definition for optional parallelism in Equation 6.25.

$$\begin{aligned}
 P &:= ((measure_p \rightarrow \overleftarrow{commPQ} \rightarrow work_p \rightarrow P) \sqcap (measure_p \rightarrow \overleftarrow{commPR} \rightarrow work_p \rightarrow P)) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((measure_q \rightarrow \overleftarrow{commPQ} \rightarrow work_q \rightarrow Q) \sqcap (measure_q \rightarrow \overleftarrow{commQR} \rightarrow work_q \rightarrow Q)) \\
 &\quad \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((measure_r \rightarrow \overleftarrow{commPR} \rightarrow work_r \rightarrow R) \sqcap (measure_r \rightarrow \overleftarrow{commQR} \rightarrow work_r \rightarrow R)) \\
 &\quad \sqcap (sleep_r \rightarrow R)
 \end{aligned} \tag{6.24}$$

$$WSN := \prod_{i=1}^3 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{commPQ, commPR, work_p, measure_p, sleep_p\})$$

$$(N_2, \alpha N_2) = (Q, \{commPQ, commQR, work_q, measure_q, sleep_q\})$$

$$(N_3, \alpha N_3) = (R, \{commPR, commQR, work_r, measure_r, sleep_r\}) \tag{6.25}$$

6.1.3.1.3 Unidirectional - Simplex

The ring topology scenarios are best described with simplex synchronisation because the communication occurs either in a clockwise or an anticlockwise direction. This scenario will be described using a single simplex communication event between the processes, resulting in unidirectional communication between the processes. For bidirectional communication, two simplex definitions with different communication directions could be combined into one. Figure 6.16 depicts uni-directional synchronisation between processes P , Q and R using a simplex synchronisation event between all of the processes in a clockwise direction. The adjacency list for this scenario is given in Listing 6.12.

(P, R)
 (Q, P)
 (R, Q)

Listing 6.12: Adjacency list for unidirectional synchronisation between processes P , Q and R using one simplex synchronisation event.

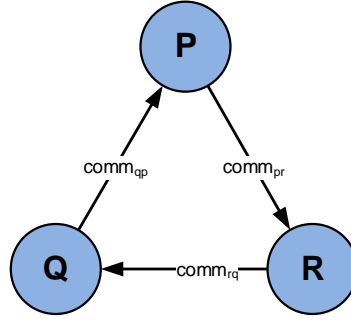


Figure 6.16: Unidirectional synchronisation between processes P , Q and R using one simplex synchronisation event.

The CSP process definitions are given in Equation 6.26 and the network definition for optional parallelism in Equation 6.27.

$$\begin{aligned}
 P &:= (measure_p \rightarrow \overrightarrow{comm_{pr}} \rightarrow work_p \rightarrow P) \square (\overleftarrow{comm_{qp}} \rightarrow work_p \rightarrow P) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= (measure_q \rightarrow \overrightarrow{comm_{qp}} \rightarrow work_q \rightarrow Q) \square (\overleftarrow{comm_{rq}} \rightarrow work_q \rightarrow Q) \\
 &\quad \sqcap (sleep_q \rightarrow Q) \\
 R &:= (measure_r \rightarrow \overrightarrow{comm_{rq}} \rightarrow work_r \rightarrow R) \square (\overleftarrow{comm_{pr}} \rightarrow work_r \rightarrow R) \\
 &\quad \sqcap (sleep_r \rightarrow R)
 \end{aligned} \tag{6.26}$$

$$WSN := \prod_{i=1}^3 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{comm_{pr}, comm_{qp}, work_p, measure_p, sleep_p\})$$

$$(N_2, \alpha N_2) = (Q, \{comm_{qp}, comm_{rq}, work_q, measure_q, sleep_q\})$$

$$(N_3, \alpha N_3) = (R, \{comm_{pr}, comm_{rq}, work_r, measure_r, sleep_r\}) \tag{6.27}$$

6.1.3.2 4-Node ring-topology

A 4-node ring topology is used as a scenario to test the behaviour of the CSP model when the network is not fully connected. In Figure 6.17 it can be noted that processes P and R do not share an edge between them and the same for processes Q and S . This means that not all node processes are able to receive all communication events. Broadcasting in this scenario has a multicast flavour, where not all of the processes receive the broadcasting event. This is per definition, hence the multicast reference.

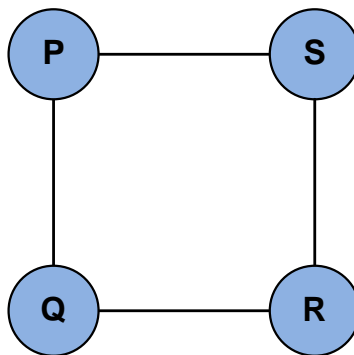


Figure 6.17: Ring topology with 4 node processes P , Q , R and S .

The 3 sub-scenarios detailing broadcasting, half-duplex bidirectional and simplex unidirectional synchronisation are detailed below.

6.1.3.2.1 Broadcasting

Figure 6.18 shows the different broadcasting approaches between processes P , Q , R and S . Each process has a unique broadcasting event to its connected processes. A broadcaster is identified and the connected processes are only able to listen and synchronise if the broadcaster performs a broadcast event. It is shown in Figure 6.18 that a directional hyper-edge exists for the broadcasting event from the particular broadcasting node. A common broadcasting event is shared between the broadcaster and the sensor nodes, but not between the sensor nodes themselves. The adjacency list is given in Listing 6.13.

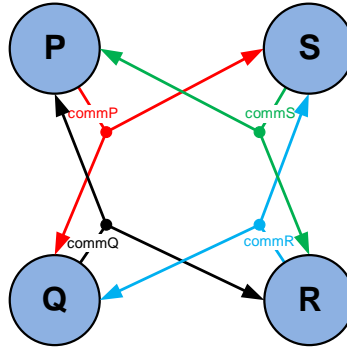


Figure 6.18: Ring topology with processes P , Q , R and S .

(P, Q, S)
(Q, P, R)
(R, Q, S)
(S, P, R)

Listing 6.13: Adjacency list for broadcasting synchronisation between processes P , Q , R and S .

The CSP process definitions for Figure 6.18 are given in Equation 6.28.

$$\begin{aligned}
 P &:= ((measure_p \rightarrow \overrightarrow{commP} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{commQ} \rightarrow work_p \rightarrow P) \\
 &\quad \sqcap (\overleftarrow{commS} \rightarrow work_p \rightarrow P)) \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((measure_q \rightarrow \overrightarrow{commQ} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{commP} \rightarrow work_q \rightarrow Q) \\
 &\quad \sqcap (\overleftarrow{commR} \rightarrow work_q \rightarrow Q)) \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((measure_r \rightarrow \overrightarrow{commR} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{commQ} \rightarrow work_r \rightarrow R) \\
 &\quad \sqcap (\overleftarrow{commS} \rightarrow work_r \rightarrow R)) \sqcap (sleep_r \rightarrow R) \\
 S &:= ((measure_s \rightarrow \overrightarrow{commS} \rightarrow work_s \rightarrow S) \sqcap (\overleftarrow{commP} \rightarrow work_s \rightarrow S) \\
 &\quad \sqcap (\overleftarrow{commR} \rightarrow work_s \rightarrow S)) \sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.28}$$

The network definition is given in Equation 6.29.

$$\begin{aligned}
 WSN &:= \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where} \\
 (N_1, \alpha N_1) &= (P, \{commP, commQ, commS, work_p, measure_p, sleep_p\}) \\
 (N_2, \alpha N_2) &= (Q, \{commP, commQ, commR, work_q, measure_q, sleep_q\}) \\
 (N_3, \alpha N_3) &= (R, \{commQ, commR, commS, work_r, measure_r, sleep_r\}) \\
 (N_4, \alpha N_4) &= (S, \{commP, commR, commS, work_s, measure_s, sleep_s\}) \quad (6.29)
 \end{aligned}$$

6.1.3.2.2 Bidirectional - Half-duplex

Figure 6.19 depicts the communication links between processes P , Q , R and S . These connections are all point-to-point, but synchronisation is done with optional parallelism. This scenario has the same structure when the general parallel operator is used. The adjacency list is given in Listing 6.14.

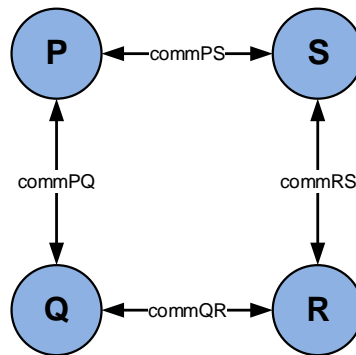


Figure 6.19: Bidirectional synchronisation between processes P , Q , R and S using half-duplex synchronisation.

```

{P,Q}
{P,S}
{Q,R}
{R,S}
  
```

Listing 6.14: Adjacency list for half-duplex synchronisation between processes P , Q , R and S .

The CSP process definitions are given in Equation 6.30 and the network definition for optional parallelism in Equation 6.31.

$$\begin{aligned}
 P &:= ((\text{measure}_p \rightarrow \overleftarrow{\text{comm}PQ} \rightarrow \text{work}_p \rightarrow P) \square (\text{measure}_p \rightarrow \overleftarrow{\text{comm}PS} \rightarrow \text{work}_p \rightarrow P)) \\
 &\quad \sqcap (\text{sleep}_p \rightarrow P) \\
 Q &:= ((\text{measure}_q \rightarrow \overleftarrow{\text{comm}PQ} \rightarrow \text{work}_q \rightarrow Q) \square (\text{measure}_q \rightarrow \overleftarrow{\text{comm}QR} \rightarrow \text{work}_q \rightarrow Q)) \\
 &\quad \sqcap (\text{sleep}_q \rightarrow Q) \\
 R &:= ((\text{measure}_r \rightarrow \overleftarrow{\text{comm}QR} \rightarrow \text{work}_r \rightarrow R) \square (\text{measure}_r \rightarrow \overleftarrow{\text{comm}RS} \rightarrow \text{work}_r \rightarrow R)) \\
 &\quad \sqcap (\text{sleep}_r \rightarrow R) \\
 S &:= ((\text{measure}_s \rightarrow \overleftarrow{\text{comm}PS} \rightarrow \text{work}_s \rightarrow S) \square (\text{measure}_s \rightarrow \overleftarrow{\text{comm}RS} \rightarrow \text{work}_s \rightarrow S)) \\
 &\quad \sqcap (\text{sleep}_s \rightarrow S)
 \end{aligned} \tag{6.30}$$

$$WSN := \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{\text{comm}PQ, \text{comm}PS, \text{work}_p, \text{measure}_p, \text{sleep}_p\})$$

$$(N_2, \alpha N_2) = (Q, \{\text{comm}PQ, \text{comm}QR, \text{work}_q, \text{measure}_q, \text{sleep}_q\})$$

$$(N_3, \alpha N_3) = (R, \{\text{comm}QR, \text{comm}RS, \text{work}_r, \text{measure}_r, \text{sleep}_r\})$$

$$(N_4, \alpha N_4) = (S, \{\text{comm}PS, \text{comm}RS, \text{work}_s, \text{measure}_s, \text{sleep}_s\}) \tag{6.31}$$

6.1.3.2.3 Unidirectional - Simplex

This scenario will be described using a single simplex communication event between the processes, resulting in unidirectional communication between the processes. Figure 6.20 depicts uni-directional synchronisation between processes P , Q , R and S using a simplex synchronisation event between all of the processes in a clockwise direction. The adjacency list for this scenario is given in Listing 6.15.

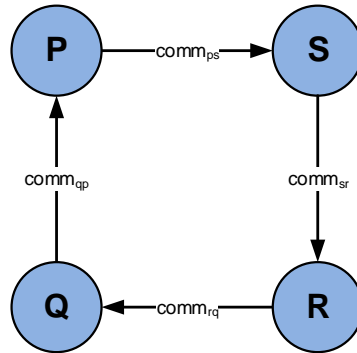


Figure 6.20: Unidirectional synchronisation between processes P , Q , R and S using one simplex synchronisation event.

(P, S)
(Q, R)
(R, Q)
(S, R)

Listing 6.15: Adjacency list for unidirectional synchronisation between processes P , Q , R and S using one simplex synchronisation event.

The CSP process definitions are given in Equation 6.32 and the network definition for optional parallelism in Equation 6.33.

$$\begin{aligned}
 P &:= ((measure_p \rightarrow \overrightarrow{comm_{ps}} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{comm_{qp}} \rightarrow work_p \rightarrow P)) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((measure_q \rightarrow \overrightarrow{comm_{qp}} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{comm_{rq}} \rightarrow work_q \rightarrow Q)) \\
 &\quad \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((measure_r \rightarrow \overrightarrow{comm_{rq}} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{comm_{sr}} \rightarrow work_r \rightarrow R)) \\
 &\quad \sqcap (sleep_r \rightarrow R) \\
 S &:= ((measure_s \rightarrow \overrightarrow{comm_{sr}} \rightarrow work_s \rightarrow S) \sqcap (\overleftarrow{comm_{ps}} \rightarrow work_s \rightarrow S)) \\
 &\quad \sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.32}$$

$$WSN := \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{comm_{ps}, comm_{qp}, work_p, measure_p, sleep_p\})$$

$$(N_2, \alpha N_2) = (Q, \{comm_{qp}, comm_{rq}, work_q, measure_q, sleep_q\})$$

$$(N_3, \alpha N_3) = (R, \{comm_{rq}, comm_{sr}, work_r, measure_r, sleep_r\})$$

$$(N_4, \alpha N_4) = (S, \{comm_{sr}, comm_{ps}, work_s, measure_s, sleep_s\}) \quad (6.33)$$

6.1.3.3 4-Node Linear Topology

A 4-node linear topology is given in Figure 6.21. This scenario is similar to the 4-node ring topology of Figure 6.17 with the exception that nodes P and S are disconnected.

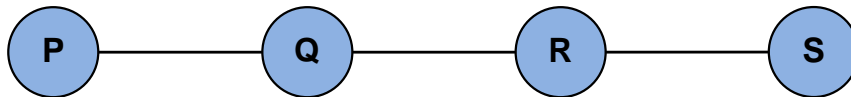


Figure 6.21: Linear topology with node processes P , Q , R and S .

The 3 sub-scenarios detailing broadcasting, half-duplex bidirectional and simplex unidirectional synchronisation are detailed below.

6.1.3.3.1 Broadcasting

Figure 6.22 shows the different broadcasting approaches between processes P , Q , R and S . Each process has a unique broadcasting event to its connected processes. A broadcaster is identified and the connected processes are only able to listen and synchronise if the broadcaster has a broadcast event. It is shown in Figure 6.22 that a directional hyper-edge exists for the broadcasting event from the particular broadcasting node. A common broadcasting event is shared between the broadcaster and the sensor nodes, but not between the sensor nodes themselves. The adjacency list is given in Listing 6.16.

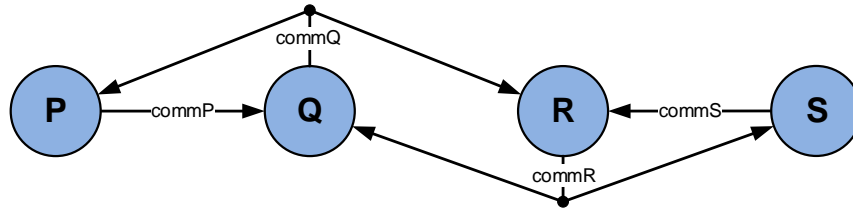


Figure 6.22: Linear topology with processes P , Q , R and S .

(P, Q)
(Q, P, R)
(R, Q, S)
(S, R)

Listing 6.16: Adjacency list for a linear topology between processes P , Q , R and S .

The CSP process definitions for Figure 6.22 are given in Equation 6.34.

$$\begin{aligned}
 P &:= ((measure_p \rightarrow \overrightarrow{commP} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{commQ} \rightarrow work_p \rightarrow P)) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((measure_q \rightarrow \overrightarrow{commQ} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{commP} \rightarrow work_q \rightarrow Q) \\
 &\quad \sqcap (\overleftarrow{commR} \rightarrow work_q \rightarrow Q)) \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((measure_r \rightarrow \overrightarrow{commR} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{commQ} \rightarrow work_r \rightarrow R) \\
 &\quad \sqcap (\overleftarrow{commS} \rightarrow work_r \rightarrow R)) \sqcap (sleep_r \rightarrow R) \\
 S &:= ((measure_s \rightarrow \overrightarrow{commS} \rightarrow work_s \rightarrow S) \sqcap (\overleftarrow{commR} \rightarrow work_s \rightarrow S)) \\
 &\quad \sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.34}$$

The network definition is given in Equation 6.35.

$$\begin{aligned}
 WSN &:= \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where} \\
 (N_1, \alpha N_1) &= (P, \{commP, commQ, work_p, measure_p, sleep_p\}) \\
 (N_2, \alpha N_2) &= (Q, \{commP, commQ, commR, work_q, measure_q, sleep_q\}) \\
 (N_3, \alpha N_3) &= (R, \{commQ, commR, commS, work_r, measure_r, sleep_r\}) \\
 (N_4, \alpha N_4) &= (S, \{commS, commR, work_s, measure_s, sleep_s\})
 \end{aligned} \tag{6.35}$$

6.1.3.3.2 Bidirectional - Half-duplex

Figure 6.23 depicts the communication links between processes P , Q , R and S . These connections are all point-to-point, but synchronisation is done with optional parallelism. This scenario has the same structure when the general parallel operator is used. The adjacency list is given in Listing 6.17.

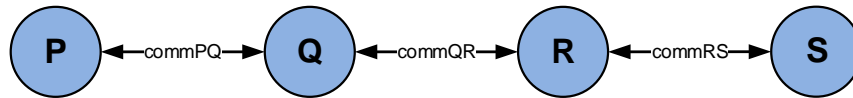


Figure 6.23: Bidirectional synchronisation between processes P , Q , R and S using half-duplex synchronisation.

$\{P, Q\}$ $\{Q, R\}$ $\{R, S\}$
--

Listing 6.17: Adjacency list for half-duplex synchronisation between processes P , Q , R and S .

The CSP process definitions are given in Equation 6.36 and the network definition for optional parallelism in Equation 6.37.

$$\begin{aligned}
 P &:= (\overleftarrow{commPQ} \rightarrow work_p \rightarrow P) \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((\overleftarrow{commPQ} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{commQR} \rightarrow work_q \rightarrow Q)) \\
 &\quad \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((\overleftarrow{commQR} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{commRS} \rightarrow work_r \rightarrow R)) \\
 &\quad \sqcap (sleep_r \rightarrow R) \\
 S &:= (\overleftarrow{commRS} \rightarrow work_s \rightarrow S) \sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.36}$$

$$WSN := \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{commPQ, work_p, measure_p, sleep_p\})$$

$$(N_2, \alpha N_2) = (Q, \{commPQ, commQR, work_q, measure_q, sleep_q\})$$

$$(N_3, \alpha N_3) = (R, \{commQR, commRS, work_r, measure_r, sleep_r\})$$

$$(N_4, \alpha N_4) = (S, \{commRS, work_s, measure_s, sleep_s\}) \tag{6.37}$$

6.1.3.3.3 Unidirectional - Simplex

This scenario will be described using a single simplex communication event between the processes, resulting in unidirectional communication between the processes. Figure 6.24 depicts uni-directional synchronisation between processes P , Q , R and S using a simplex synchronisation event between all of the processes in a direction from left to right. The adjacency list for this scenario is given in Listing 6.18.

(P, Q)
 (Q, R)
 (R, S)

Listing 6.18: Adjacency list for unidirectional synchronisation between processes P , Q , R and S using one simplex synchronisation event.

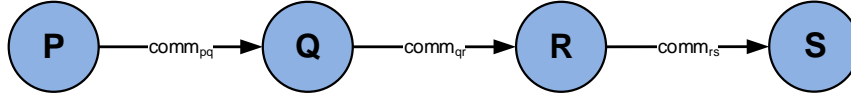


Figure 6.24: Unidirectional synchronisation between processes P , Q , R and S using one simplex synchronisation event.

The CSP process definitions are given in Equation 6.38 and the network definition for optional parallelism in Equation 6.39.

$$\begin{aligned}
 P &:= (measure_p \rightarrow \overrightarrow{comm_{pq}} \rightarrow work_p \rightarrow P) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= (measure_q \rightarrow \overrightarrow{comm_{qr}} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{comm_{pq}} \rightarrow work_q \rightarrow Q) \\
 &\quad \sqcap (sleep_q \rightarrow Q) \\
 R &:= (measure_r \rightarrow \overrightarrow{comm_{rs}} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{comm_{qr}} \rightarrow work_r \rightarrow R) \\
 &\quad \sqcap (sleep_r \rightarrow R) \\
 S &:= (\overleftarrow{comm_{rs}} \rightarrow work_s \rightarrow S) \\
 &\quad \sqcap (sleep_s \rightarrow S)
 \end{aligned} \tag{6.38}$$

$$WSN := \prod_{i=1}^4 (N_i, \alpha N_i), \text{ where}$$

$$(N_1, \alpha N_1) = (P, \{comm_{pq}, work_p, measure_p, sleep_p\})$$

$$(N_2, \alpha N_2) = (Q, \{comm_{pq}, comm_{qr}, work_q, measure_q, sleep_q\})$$

$$(N_3, \alpha N_3) = (R, \{comm_{rs}, comm_{qr}, work_r, measure_r, sleep_r\})$$

$$(N_4, \alpha N_4) = (S, \{comm_{rs}, work_s, measure_s, sleep_s\}) \tag{6.39}$$

6.1.4 Tree

In a tree topology, the leaf nodes send their data to their parent nodes, which in turn send the aggregated information to their own parent nodes. Using this data aggregation approach, data is passed from the leaf nodes to the root node, which typically acts as the sink node. Tree topologies attempt to avoid flooding and unicast can be used over broadcast. Optional parallelism is used to model the point-to-point links where a leaf node can transmit a data message even if its parent node is not ready to receive data. This allows the leaf node to still be able to respond to its leaf nodes' data transmissions. Stated differently, the use of optional parallelism to model tree topologies in CSP covers cases where the whole branch blocks if a parent node cannot service a data transmission request at that specific moment. Optional parallelism is therefore used for the point-to-point (unicast) links. From a modelling perspective, communication from the root node to the leaf nodes will be done with broadcasting.

A 7-node tree-topology is given in Figure 6.25.

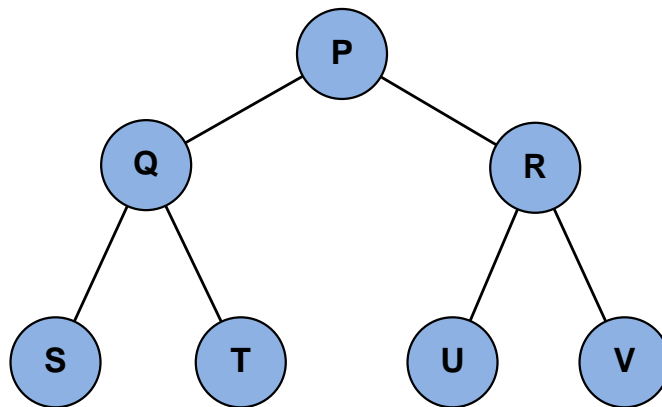


Figure 6.25: Tree topology with processes P , Q , R , S , T , U and V .

The 3 sub-scenarios detailing broadcasting, half-duplex bidirectional and simplex unidirectional synchronisation are detailed below.

6.1.4.1 Broadcasting

Figure 6.26 shows the broadcasting approach between processes P , Q , R , S , T , U and V . Communication takes place in one direction from the root node down to the leaf node. Each process has a unique broadcasting event to its connected leaf nodes. A broadcaster is identified and the connected leaf node processes are only able to listen and synchronise if the broadcaster has a broadcast event. It is shown in Figure 6.26 that a directional hyper-edge exists for the broadcasting event from the particular broadcasting node. A common broadcasting event is shared between the broadcaster and the sensor nodes, but not between the sensor nodes themselves. The adjacency list is given in Listing 6.19.

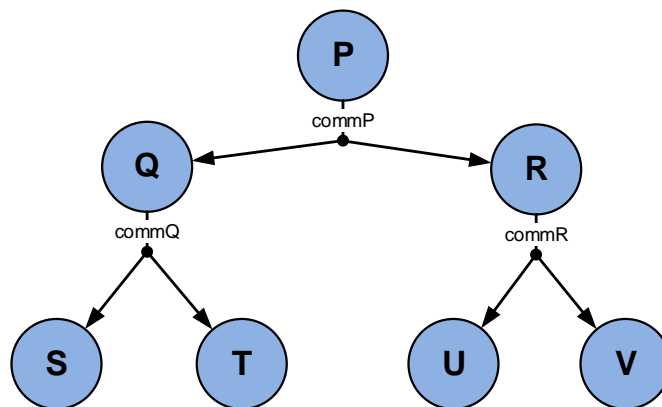


Figure 6.26: Tree topology with processes P , Q , R , S , T , U and V .

```

(P, Q, R)
(Q, S, T)
(R, U, V)
  
```

Listing 6.19: Adjacency list for a tree topology between processes P , Q , R , S , T , U and V .

The CSP process definitions for Figure 6.26 are given in Equation 6.40.

$$\begin{aligned}
 P &:= (measure_p \rightarrow \overrightarrow{commP} \rightarrow work_p \rightarrow P) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((measure_q \rightarrow \overrightarrow{commQ} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{commP} \rightarrow work_q \rightarrow Q)) \\
 &\quad \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((measure_r \rightarrow \overrightarrow{commR} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{commP} \rightarrow work_r \rightarrow R)) \\
 &\quad \sqcap (sleep_r \rightarrow R) \\
 S &:= (\overleftarrow{commQ} \rightarrow work_s \rightarrow S) \sqcap (sleep_s \rightarrow S) \\
 T &:= (\overleftarrow{commQ} \rightarrow work_t \rightarrow T) \sqcap (sleep_t \rightarrow T) \\
 U &:= (\overleftarrow{commR} \rightarrow work_u \rightarrow U) \sqcap (sleep_u \rightarrow U) \\
 V &:= (\overleftarrow{commR} \rightarrow work_v \rightarrow V) \sqcap (sleep_v \rightarrow V)
 \end{aligned} \tag{6.40}$$

The network definition is given in Equation 6.41.

$$\begin{aligned}
 WSN &:= \prod_{i=1}^7 (N_i, \alpha N_i) , \text{ where} \\
 (N_1, \alpha N_1) &= (P, \{commP, work_p, measure_p, sleep_p\}) \\
 (N_2, \alpha N_2) &= (Q, \{commP, commQ, work_q, measure_q, sleep_q\}) \\
 (N_3, \alpha N_3) &= (R, \{commP, commR, work_r, measure_r, sleep_r\}) \\
 (N_4, \alpha N_4) &= (S, \{commQ, work_s, sleep_s\}) \\
 (N_5, \alpha N_5) &= (T, \{commQ, work_t, sleep_t\}) \\
 (N_6, \alpha N_6) &= (U, \{commR, work_u, sleep_u\}) \\
 (N_7, \alpha N_7) &= (V, \{commR, work_v, sleep_v\})
 \end{aligned} \tag{6.41}$$

6.1.4.2 Bidirectional - Half-duplex

Figure 6.27 depicts the communication links between processes P , Q , R , S , T , U and V . These connections are all point-to-point, but synchronisation is done with optional parallelism. This scenario has the same structure when the general parallel operator is used. The adjacency list is given in Listing 6.20.

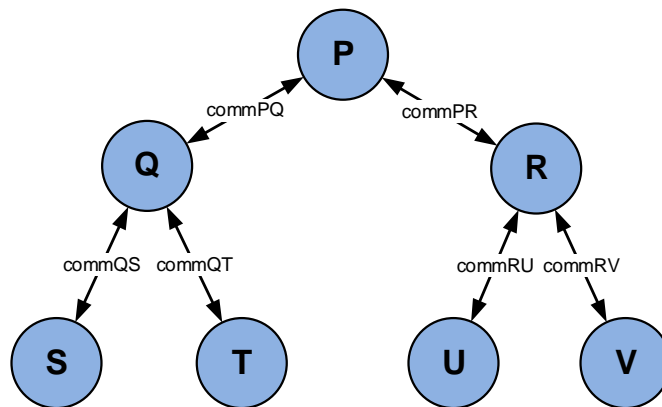


Figure 6.27: Bidirectional synchronisation between processes P , Q , R , S , T , U and V using half-duplex synchronisation.

{P,Q}
{P,R}
{Q,S}
{Q,T}
{R,U}
{R,V}

Listing 6.20: Adjacency list for half-duplex synchronisation between processes P , Q , R , S , T , U and V .

The CSP process definitions are given in Equation 6.42 and the network definition for optional parallelism in Equation 6.43.

$$\begin{aligned}
 P &:= ((measure_p \rightarrow \overleftarrow{commPQ} \rightarrow work_p \rightarrow P) \sqcap (measure_p \rightarrow \overleftarrow{commPR} \rightarrow work_p \rightarrow P)) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((measure_q \rightarrow \overleftarrow{commPQ} \rightarrow work_q \rightarrow Q) \sqcap (measure_q \rightarrow \overleftarrow{commQS} \rightarrow work_q \rightarrow Q)) \\
 &\quad \sqcap (measure_q \rightarrow \overleftarrow{commQT} \rightarrow work_q \rightarrow Q) \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((measure_r \rightarrow \overleftarrow{commPR} \rightarrow work_r \rightarrow R) \sqcap (measure_r \rightarrow \overleftarrow{commRU} \rightarrow work_r \rightarrow R)) \\
 &\quad \sqcap (measure_r \rightarrow \overleftarrow{commRV} \rightarrow work_r \rightarrow R) \sqcap (sleep_r \rightarrow R) \\
 S &:= (measure_s \rightarrow \overleftarrow{commQS} \rightarrow work_s \rightarrow S) \sqcap (sleep_s \rightarrow S) \\
 T &:= (measure_t \rightarrow \overleftarrow{commQT} \rightarrow work_t \rightarrow T) \sqcap (sleep_t \rightarrow T) \\
 U &:= (measure_u \rightarrow \overleftarrow{commRU} \rightarrow work_u \rightarrow U) \sqcap (sleep_u \rightarrow U) \\
 V &:= (measure_v \rightarrow \overleftarrow{commRV} \rightarrow work_v \rightarrow V) \sqcap (sleep_v \rightarrow V)
 \end{aligned} \tag{6.42}$$

$$WSN := \prod_{i=1}^7 (N_i, \alpha N_i), \text{ where}$$

$$\begin{aligned}
 (N_1, \alpha N_1) &= (P, \{commPQ, commPR, work_p, measure_p, sleep_p\}) \\
 (N_2, \alpha N_2) &= (Q, \{commPQ, commQS, commQT, work_q, measure_q, sleep_q\}) \\
 (N_3, \alpha N_3) &= (R, \{commPR, commRU, commRV, work_r, measure_r, sleep_r\}) \\
 (N_4, \alpha N_4) &= (S, \{commQS, work_s, measure_s, sleep_s\}) \\
 (N_5, \alpha N_5) &= (S, \{commQT, work_s, measure_s, sleep_s\}) \\
 (N_6, \alpha N_6) &= (S, \{commRU, work_s, measure_s, sleep_s\}) \\
 (N_7, \alpha N_7) &= (S, \{commRV, work_s, measure_s, sleep_s\})
 \end{aligned} \tag{6.43}$$

6.1.4.3 Unidirectional - Simplex

This scenario will be described using a single simplex communication event between the processes, resulting in unidirectional communication between the processes from the leaf nodes back to the root node. Figure 6.28 depicts uni-directional synchronisation between processes P , Q , R , S , T , U and V using a simplex synchronisation event between all of the processes in a direction from the leaf nodes to the root node (bottom to top). The adjacency list for this scenario is given in Listing 6.21.

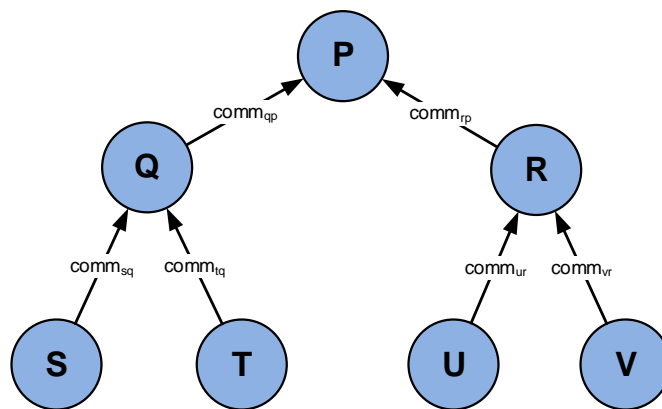


Figure 6.28: Unidirectional synchronisation between processes P , Q , R , S , T , U and V using one simplex synchronisation event.

```
(Q,P)
(R,P)
(S,Q)
(T,Q)
(U,R)
(V,R)
```

Listing 6.21: Adjacency list for unidirectional synchronisation between processes P , Q , R , S , T , U and V using one simplex synchronisation event.

The CSP process definitions are given in Equation 6.44 and the network definition for optional parallelism in Equation 6.45.

$$\begin{aligned}
 P &:= ((\overleftarrow{comm}_{qp} \rightarrow work_p \rightarrow P) \sqcap (\overleftarrow{comm}_{rp} \rightarrow work_p \rightarrow P)) \\
 &\quad \sqcap (sleep_p \rightarrow P) \\
 Q &:= ((measure_q \rightarrow \overrightarrow{comm}_{qp} \rightarrow work_q \rightarrow Q) \sqcap (\overleftarrow{comm}_{sq} \rightarrow work_q \rightarrow Q) \\
 &\quad \sqcap (\overleftarrow{comm}_{tq} \rightarrow work_q \rightarrow Q)) \sqcap (sleep_q \rightarrow Q) \\
 R &:= ((measure_r \rightarrow \overrightarrow{comm}_{rp} \rightarrow work_r \rightarrow R) \sqcap (\overleftarrow{comm}_{ur} \rightarrow work_r \rightarrow R) \\
 &\quad \sqcap (\overleftarrow{comm}_{vr} \rightarrow work_r \rightarrow R)) \sqcap (sleep_r \rightarrow R) \\
 S &:= (measure_s \rightarrow \overrightarrow{comm}_{sq} \rightarrow work_s \rightarrow S) \sqcap (sleep_s \rightarrow S) \\
 T &:= (measure_t \rightarrow \overrightarrow{comm}_{tq} \rightarrow work_t \rightarrow T) \sqcap (sleep_t \rightarrow T) \\
 U &:= (measure_u \rightarrow \overrightarrow{comm}_{ur} \rightarrow work_u \rightarrow U) \sqcap (sleep_u \rightarrow U) \\
 V &:= (measure_v \rightarrow \overrightarrow{comm}_{vr} \rightarrow work_v \rightarrow V) \sqcap (sleep_v \rightarrow V)
 \end{aligned} \tag{6.44}$$

$$WSN := \prod_{i=1}^7 (N_i, \alpha N_i), \text{ where}$$

$$\begin{aligned}
 (N_1, \alpha N_1) &= (P, \{comm_{qp}, comm_{rp}, work_p, sleep_p\}) \\
 (N_2, \alpha N_2) &= (Q, \{comm_{qp}, comm_{sq}, comm_{tq}, work_q, measure_q, sleep_q\}) \\
 (N_3, \alpha N_3) &= (R, \{comm_{rp}, comm_{ur}, comm_{vr}, work_r, measure_r, sleep_r\}) \\
 (N_4, \alpha N_4) &= (S, \{comm_{sq}, work_s, measure_s, sleep_s\}) \\
 (N_5, \alpha N_5) &= (T, \{comm_{tq}, work_t, measure_t, sleep_t\}) \\
 (N_6, \alpha N_6) &= (U, \{comm_{ur}, work_u, measure_u, sleep_u\}) \\
 (N_7, \alpha N_7) &= (V, \{comm_{vr}, work_v, measure_v, sleep_v\})
 \end{aligned} \tag{6.45}$$

6.2 CHAPTER SUMMARY

This chapter defined some common WSN topologies and typical CSP modelling scenarios to be tested with the new optional parallel operators. The results of the tests to be done will indicate if optional parallelism can be used to model these WSNs as well as if there exists a trace relation between the two optional parallel operators. The sensor nodes were modelled as processes, defining their basic operational behaviour. The common structure for a sensor node's behaviour is to measure a phenomenon, do some processing on the measured phenomenon and communicate the processed data. In a typical practical scenario, a sensor node will measure a phenomenon by sampling its Analog-to-Digital Converter (ADC), process the reading by adding a conversion factor to it, filter out possible noise and transmit the measurement to its neighbours. This structure was adhered to as far as possible for the scenarios.

With the use of more complex topologies, the direction of the data-flow was taken into account. Although the CSP definitions in this dissertation are data-independent, the directional flow of messages had to be mentioned. Optional parallelism implicitly defines directional synchronisation due to its broadcasting principles, which was used to model the data-flow direction on a higher level.

Three sub-scenarios with different synchronisation approaches were attempted for each of the scenarios, broadcasting, bidirectional half-duplex and bidirectional simplex. Some of the scenarios had their bidirectional simplex sub-scenarios replaced by unidirectional simplex scenarios as to adhere to the goal of what the specific topology is trying to achieve.

Process definitions for the more complex topologies were given in indexed notation which included the alphabet of each node process definition, the same notation as used in alphabetised parallelism.

The optional parallel translation defined in Chapter 4 will be tested with the use of the scenarios of this chapter and the software tools implemented in Chapter 5. Chapter 7 will detail the test results in terms of CSP trace refinement and deadlock freedom of the WSN scenarios presented in this chapter.

CHAPTER 7

TESTING OPTIONAL PARALLELISM IN THE TRACES DOMAIN

The scenarios of Chapter 6 are tested with software tools to determine if the classical CSP conversion of optional parallelism of Chapter 4, $OptPar^T$, is a sufficient solution to $OptPar$. The tests are performed with the use of the new tools developed, detailed in Chapter 5, and the CSP model-checker ProCSP. The tests are done on an exploratory basis to empirically determine if common WSN system definitions, based on their graph structures or topologies, can be modelled with the optional parallel operator. The scenarios of Chapter 6 were chosen to cover the most commonly used WSN topologies. This is by no means an exhaustive test where it can be stated that there exists a trace relation between $OptPar^T$ and $OptPar$ for *all* possible examples. These exploratory tests aim to cover simple, but expandable, scenarios of common WSN topologies used in practice.

The traces of the WSN systems using $OptPar$ and $OptPar^T$ are compared to determine if a trace refinement relation exist, which will show if the classical CSP translation can be used in the place of the optional parallel operator. This is done because $OptPar$ is yet to be included in current model-checker implementations and if the applicability of optional parallelism can be shown, it can be argued that the optional parallel operator should be included in model-checking tool implementations.

Trace refinement between $OptPar$ and $OptPar^T$ is the ultimate test and the main result of this dissertation. If this can be determined, it can be argued that the translation is a suitable solution to model the behaviour of optional parallelism using classical CSP operators in the traces domain. This allows the trace semantics of the classical CSP operators used in

$OptPar^T$ to be inherited. Defining and proving the trace semantics of optional parallelism from first principles is a difficult task and this approach makes it easier to adopt the optional parallel operator in existing model-checkers because the trace semantics are composed of the existing CSP operators' trace semantics.

7.1 TEST DEFINITION

The WSN scenarios of Chapter 6 are tested for various properties. As previously mentioned, these systems need to be translated from system definitions using the optional parallel operator to system definitions using only the classical CSP operators. The conversion is done with the OptoCSP tool by giving the process and system definitions of Chapter 6 in CSP_M format as detailed in §5.1.1. The tests are done with the help of software tools, the two new tools OptoCSP and OpTrace as well as a third party tool, ProCSP. These converted systems need to be tested for the following:

- Deadlock Freedom
- Trace Refinement

OpTrace and ProCSP were mostly used in the deadlock and trace refinement tests. The adjacency lists for each of the test scenarios of Chapter 6 were given as a text file input to OpTrace where an internal set of processes and channels were constructed. The traces of these processes were computed by clicking the "Generate Traces" button on the user interface of OpTrace, after which the "Build Model" and "Trace Refinement" buttons builds the CSP model of the input text file and performs a trace refinement check by invoking ProCSP respectively. The result on the user interface of OpTrace and ProCSP were used to evaluate the pass criteria.

7.1.1 Deadlock freedom

Due to the networks not being triple disjoint, deadlock cannot be theoretically dismissed. It is therefore required to perform a deadlock analysis on each scenario with the use of ProCSP. This is done with the deadlock analysis function of ProCSP, given the CSP_M representation of the WSN as input. Some of the scenarios were chosen to create possible circular wait

deadlock scenarios in order to empirically check the behaviour of optional parallelism in these scenarios. As a first test criterion, it is required that the system definitions using the classical CSP translation of optional parallelism are all deadlock free. It is argued that optional parallelism allows processes to synchronise when they are able to, and to proceed independently if they are not, effectively eliminating the possibility of deadlock. This test will confirm if this is true for the selected test scenarios.

7.1.2 Trace refinement

The traces are computed with the OpTrace tool where the adjacency list of the WSN topology is given as input. OpTrace generates a system definition and performs trace refinement checks between each of the generated traces and the system definition. These trace refinement checks are performed with the help of ProCSP, where the traces are generated by OpTrace and the refinement checks done by ProCSP. If all the generated traces are possible traces of the system, it is concluded that there exists a trace refinement between $OptPar$ and $OptPar^T$. This will show that the optional parallel operator's behaviour in the traces domain can be defined using classical CSP operators.

7.1.3 Pass criteria

A test scenario will pass if it satisfies all the test criteria. The following pass criteria will be evaluated to determine if a test scenario has passed.

- **Deadlock Freedom** - The output system generated by OpTrace should be deadlock free. This is performed by ProCSP when the "Model Check" option is chosen from the option menu. ProCSP traverses all possible unique states of the input system and gives an output with details about the number of states and transitions as well as possible deadlock conditions. The user is clearly notified if a deadlock condition is found, showing the traces leading up to the deadlock condition. A pass condition will be marked if ProCSP concludes that the system is deadlock free.
- **Trace Refinement** - The trace refinement check can be done in two ways. First, OpTrace can invoke ProCSP in the background to perform CSP assertion checks on all the generated traces and give output to the user on the number of traces checked, the

number of passed assertions, the number of failed assertions and the number of errors. Secondly, the trace refinement checks can be performed using ProCSP by selecting the "Check CSP-M Assertions" function from the option menu. The test will pass if each of the generated traces is a possible trace of the input system. This is shown by OpTrace when the number of pass traces equals the number of checked traces and no errors or failed states are present. Using ProCSP, a pass condition is found if all CSP trace assertion checks are marked with a ✓ mark.

7.2 LIMITATIONS

7.2.1 Problems encountered

While the tests were executed, various software limitations were encountered. When working with large networks, memory constraints started to get problematic. OpTrace ran into the *state space explosion* problem. The number of traces generated for the test scenarios grew exponentially, eventually resulting in *out of memory* errors. This is contributed to the interleaving nature of the optional parallel operator's behaviour in the traces domain where any number of processes can opt out of synchronisation events, and all permutations have to be accounted for. The number of recursive function calls also had an effect on the amount of memory required, which increased significantly with the number of process definitions used. *Stack overflow* errors occurred frequently for larger systems where multiple recursive function calls were made.

The other limitations were encountered by ProCSP. The structure of the CSP network and the different synchronisation mechanisms influenced ProCSP's ability to model-check it. For system definitions which were too complex in terms of the number of processes and channel artefacts, *stack overflow* and *out of memory* errors occurred. Furthermore, if the number of trace refinement tests to be done exceeded 10,000, the CSP_M parser of ProCSP encountered a *stack overflow* error.

7.2.2 Solutions

Some system simplifications were made to alleviate the problems, but the same memory issues were eventually encountered by OpTrace as well as ProCSP.

OpTrace was modified to use more efficient datasets to store the traces and the details of the traces were reduced to only contain the essential information. The use of references instead of duplicating the data significantly reduced the amount of data kept in memory for each trace. This is known as *pass by reference* in contrast to *pass by value*, where the former only keeps a memory reference to the data and the latter a duplication of the data. The amount of memory required for the multiple recursion calls could unfortunately not be solved as recursion seemed to be the only implementation strategy viable for the implementation of the optional parallel trace generator of OpTrace. Increasing the stack size of the software reduced the frequency of the *stack overflow* errors, but they eventually started to occur once larger systems were tested.

To alleviate the memory errors of ProCSP, the simplifications made on the CSP system definitions was to hide all the internal independent traces of each node process. This had the result that the system definitions only contained traces of synchronisation events. It was decided that this is sufficient because the synchronising events are the only points of interest. For ProCSP's CSP_M parser running out of memory on large trace sets, the number of trace refinement checks were split into blocks of 10,000 tests, with a model file generated for each block of trace tests. There was no means to increase the stack or the total allowed memory to be used by ProCSP and therefore the memory issues could not be resolved completely.

7.2.3 Ignored metrics

With the number of theoretically generated traces having such significance on the examples that could be tested, this is also given in the test results. Other test metrics have been left out as they do not contribute to the analysis.

- **Memory** - Memory usage is implementation specific. Although the memory limitations limit the number of nodes in the test scenarios, it is a secondary effect of the vast number of states being generated. Adding more system memory to the test hardware will not




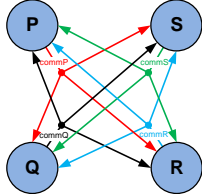
have an effect as the memory requirement grows at the same exponential rate as the number of states. This does not add any value to see if the theoretical traces refines the translation.

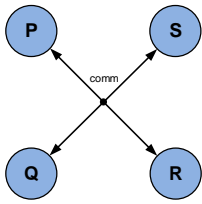
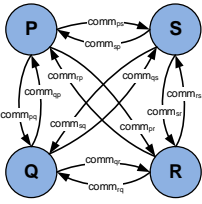
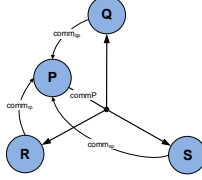
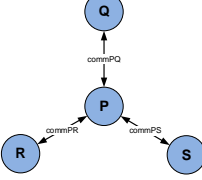
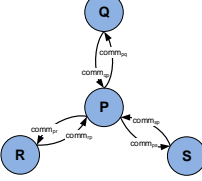
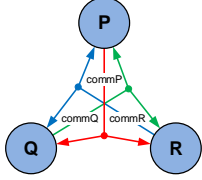
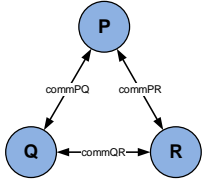
- **Execution Time** - The execution time needed for model-checking increases as the complexity of the system model increases. Although it is an indication on the complexity of the model, it does not add any value to see if the theoretical traces refines the translation.

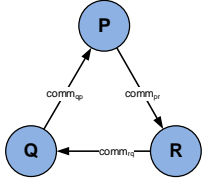
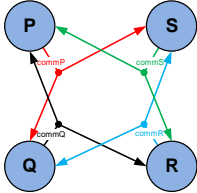
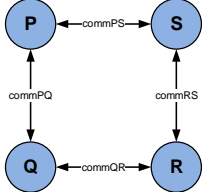
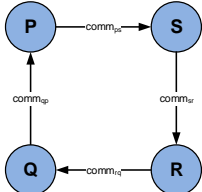
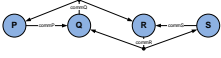
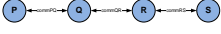

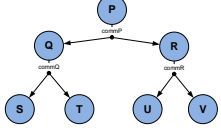
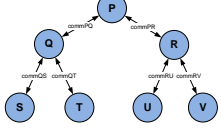
7.3 RESULTS

The results of the analysis of the WSN topology scenarios of Chapter 6 are given in Table 7.1. The following abbreviations and notation is used:

- DF** Deadlock free
ND Not determined
TR Trace refinement
 ✓ Test passed
 × Test failed

Topology	Sync	Description Reference	Diagram	CSP _M Model Reference	Traces	DF	TR
Flat, point-to-point	Broadcasting	6.1.1.1.1		B.1.1.1	3	✓	✓
	Bidirectional Half-duplex	6.1.1.1.2		B.1.1.2	7	✓	✓
	Bidirectional Simplex	6.1.1.1.3		B.1.1.3	7	✓	✓
Mesh Fully Connected	Broadcasting	6.1.1.2.1		B.1.2.1	509	ND	✓

Topology	Sync	Description Reference	Diagram	CSP _M Model Reference	Traces	DF	TR
	Bidirectional Half-duplex	6.1.1.2.2		B.1.2.2	234	✓	✓
	Bidirectional Simplex	6.1.1.2.3		B.1.2.3	4309	ND	✓
Cluster Star	Broadcasting	6.1.2.1		B.2.1.1	215	✓	✓
	Bidirectional Half-duplex	6.1.2.2		B.2.1.2	277	✓	✓
	Bidirectional Simplex	6.1.2.3		B.2.1.3	277	✓	✓
Chain 3 Node Ring	Broadcasting	6.1.3.1.1		B.3.1.1	46	✓	✓
	Bidirectional Half-duplex	6.1.3.1.2		B.3.1.2	121	✓	✓

Topology	Sync	Description Reference	Diagram	CSP _M Model Reference	Traces	DF	TR
	Unidirectional Simplex	6.1.3.1.3		B.3.1.3	28	✓	✓
Chain 4 Node Ring	Broadcasting	6.1.3.2.1		B.3.2.1	305	✓	✓
	Bidirectional Half-duplex	6.1.3.2.2		B.3.2.2	1169	✓	✓
	Unidirectional Simplex	6.1.3.2.3		B.3.2.3	153	✓	✓
Chain 4 Node Line	Broadcasting	6.1.3.3.1		B.3.3.1	223	✓	✓
	Bidirectional Half-duplex	6.1.3.3.2		B.3.3.2	399	✓	✓
	Unidirectional Simplex	6.1.3.3.3		B.3.3.3	49	✓	✓
Tree	Broadcasting	6.1.4.1		B.4.1.1	1626	✓	✓
	Bidirectional Half-duplex	6.1.4.2		B.4.1.2	536567	✓	✓

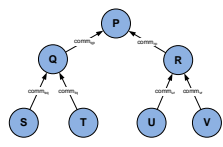
Topology	Sync	Description Reference	Diagram	CSP _M Model Reference	Traces	DF	TR
	Unidirectional Simplex	6.1.4.3		B.4.1.3	23377	✓	✓

Table 7.1: Result summary of topology tests.

7.4 DISCUSSION OF RESULTS

The software limitations of both OpTrace and ProCSP restricted the network sizes of the test scenarios. WSN simulations are often done on hundreds of nodes. This, however, is not possible with CSP models as each possible state of the network is computed based on the laws and semantics of CSP, much of which are defined in [2]. The tests which were not possible to be executed were also detailed for completeness. FDR3 was considered to be used instead of ProCSP, due to its improved simplifications and better memory management, but it was decided that it will not add value as only two scenarios could not be proven deadlock free, but all scenarios could successfully be checked for trace refinement. It was also too late in the project to change the CSP model-checker to FDR3 as it would have required software design changes as well as re-testing all previously tested scenarios.

7.4.1 Traces

The two tests which had no result on their deadlock freeness tests was due to ProCSP running out of memory. Both of these tests were from the fully connected mesh topology group, using broadcasting and bidirectional simplex channels, where all processes are connected to each other. It is clear from their network graphs that their CSP definitions would be complex, with many processes and channel artefacts. When the broadcasting and bidirectional simplex scenarios of the fully connected mesh topology are compared to their bidirectional half-duplex counterpart, it can be seen that the number of traces are a lot less, in contrast with all the other scenarios where the number of traces of bidirectional half-duplex channels are the same as the bidirectional simplex channels. This is due to the fact that this is the best example to

illustrate optional parallelism. In the bidirectional half-duplex scenario, there is one common synchronisation event *comm* between the processes, and each process has an equal chance to either be a *transmitter* or *receiver*, as well as to opt out of being one of the receiving processes. This example shows that the commonly used generalised parallelism examples in CSP, where all processes share the same synchronisation event, can be modelled with relatively low complexity using optional parallelism as only one channel artefact is added per process.

In each of the scenarios, except for the fully connected mesh topology scenario, the number of traces of the bidirectional half-duplex and bidirectional simplex are the same. Further analysis in ProCSP indicated that the number of state transitions and the number of states differ for the cases where the possible traces were the same. The bidirectional simplex scenarios had more than double the amount of states and state transitions, which were expected because the number of channel artefacts are doubled in the bidirectional simplex cases. For this reason, some of the scenarios were modified to have unidirectional simplex definitions to see if some other trace related conclusions could be made. In these scenarios, the amount of possible traces were significantly reduced due to only half the amount of channel artefacts used. The number of states and state transitions followed this trend accordingly.

When the 4-node ring and 4-node chain scenarios are compared to each other, it can be seen that the addition of the extra channel artefacts to convert from a line to a ring topology greatly increased the number of traces. It was expected that the number of traces would only increase by a few traces as the channels added only influenced the two of the processes. In the bidirectional and unidirectional scenarios of the 4-node line topology, the increase in the number of traces was almost threefold more when the topology was converted to a chain topology. It was later found, by inspection of the traces, that many new permutations of the traces were found due to the additional channel artefacts.

The tree topology had the most amount of traces, significantly more than all the other scenarios. This is due to these scenarios having more processes and therefore more channel artefacts connected to them.

7.4.2 Deadlock

The two tests on which no verdict could be given regarding their deadlock freeness were not marked as failed, but only as not testable because more refined model-checkers could be used for analysis. Simulation runs where the node processes were simulated in independent software threads gave initial insight to whether a complex network configuration could possibly be testable. After a successfully completed simulation run, it could be concluded that there could possibly exist a trace refinement and the system could be checked for deadlock freeness. Trace discrepancies and deadlock conditions are easily shown with the simulator where time-out errors are shown to the user. For the tests performed in this chapter, no possible deadlock conditions or trace discrepancies have been found with the simulation runs.

It has been found that all tests which did not suffer from state space explosion passed the deadlock test criterion. They were all deadlock free, even in the cases where possible circular-wait conditions could have occurred like the fully connected mesh and ring topologies.

7.4.3 Trace refinement

All the tests' mathematically computed traces with the use of OpTrace were refinements of their optional parallel translation, confirming the specification and implementation relation of Equation 5.1. It can be concluded, from the proof tests, that $OptPar^T$ is refined by $OptPar$ for these specific test scenarios. These test scenarios serve as a good result set as more complex topologies can be built from a combination of the tested topologies. Currently, nothing can be said about a possible trace refinement relation where $OptPar^T$ refines $OptPar$. To test a refinement relation as Equation 5.1 where the *SPEC* and *IMPL* are swapped requires a CSP model-checker to have $OptPar$ implemented. Then, the traces of systems using $OptPar^T$ can be tested for refinement against systems containing $OptPar$ with the use of a CSP model-checker. If a trace refinement could be found in both directions where $OptPar^T$ and $OptPar$ both refine each other, it could be concluded that $OptPar^T$ is an *exact* translation of $OptPar$.

CHAPTER 8

CONCLUSION AND OUTLOOK

Communicating processes under parallelism in CSP require them to synchronise jointly. It is often found in real-world scenarios that this is not possible. A WSN is a textbook example of a scenario where a node communicates with its neighbours, and not all neighbouring nodes are required to respond. In some cases the neighbouring nodes are not able to respond due to the limiting factors of WSNs, described in §2.1. This is typically seen during broadcasting, which is not natively supported in CSP. The introduction of optional parallelism addresses this shortfall in the current CSP language. The notion of broadcasting nodes can easily be modelled as CSP processes using the optional parallel operator. This allows for partial synchronisation between processes which are able to synchronise and allows processes to opt out of synchronisation when they are not ready to do so.

This dissertation explored the initial work on optional parallelism of [17] (referred to as *OptPar*) by modelling its behaviour with classical CSP operators (referred to as *OptPar^T*). This was done by adding a CSP channel artefact to each of the processes' definitions. The channel artefacts, one for each synchronising event, were added under alphabetised parallelism to the corresponding process. The processes effectively had their external synchronisation events hidden to only be communicated with its attached channel artefacts. The channel artefacts orchestrate the communication with each other, so that when the traces of interest of the system are observed, the traces of the step laws of *OptPar* are represented. This addition of the channel artefacts to the communicating processes serve as a translation from CSP systems defined with the optional parallel operator of [17] to CSP systems using only classical CSP operators. It has been found that various communication properties can be modelled with the atomic channel artefacts. This allows for broadcasting, simplex, half-

duplex and full-duplex communication per channel to be modelled.

$OptPar^T$ was tested for trace refinement against the step laws of $OptPar$ [17] by means of OpTrace, a newly developed CSP trace generator, and the model-checking tool ProCSP, an extension of the ProB tool to support CSP. Various WSN graph topologies were modelled in CSP from real-world applications. The traces of these systems were computed by OpTrace's optional parallel operator implementation. The traces were checked for trace refinement against the translated WSN CSP model with the use of ProCSP. All the tested scenarios passed the trace refinement checks, meaning that all the generated traces can be observed from the system definitions. All the scenarios were proven to be deadlock free with the deadlock analysis function of ProCSP, except for two as their systems were too complex for ProCSP to analyse.

With the software limitations encountered, not all scenarios could be tested for deadlock freedom. It was still useful to do as the concept of the translation of optional parallelism could be tested, albeit for small scenarios. ProCSP is not a production model-checker such as FDR, but was sufficient in the task to prove the concept of translating optional parallelism to a combination of classical CSP operators.

The hypothesis of §1.3 can be evaluated with the results of the test scenarios. It is restated below:

“Optional parallelism can be defined using classical CSP operators to have the same behaviour in the traces domain.”

The hypothesis is partially accepted. It is true for the WSN scenarios tested, but it can not be concluded that it will hold true for all examples. This is a topic for future research.

The research contribution of this dissertation is the in depth study of optional parallelism and testing its applicability to model WSNs. The translation of the behaviour of optional parallelism in the traces domain and successfully testing it against the optional parallel definition of [17] serves as a good result on the topic of optional parallelism. The overall goal is to accept the optional parallel operator as a CSP operator such that it can be implemented in model-checking tools. New tools, OptoCSP and OpTrace were developed to aid in the task of testing various network scenarios. These tools can be used in further research on the topic and

assist in the implementation of optional parallelism in existing CSP model-checkers.

The results show that optional parallelism provides a means to check network parameters without delving too deep into the protocol specifics, although it could be useful, especially to model the broadcasting parts of current WSN communication and routing protocols as well as collision detection. Modelling WSNs in CSP is used to highlight different parameters from conventional WSN simulators [114] and [115], such as the absence of system failures and deadlock. Optional parallelism aids in this task as it is more applicable to real-world WSN scenarios where nodes can run out of power resources and stop engaging in communication. The solution of this dissertation provides a means to thoroughly check WSN behaviour where all possible states of the network is computed and tested against its specification. With the notion of time not being included in classical CSP, the models are not concerned *when* a state is reached, but rather *if* a state is reached.

It would have been a good result if the laws of the operational semantics of optional parallelism given in [17] could be proven mathematically. The approach used in this dissertation was to translate the operational semantics of optional parallelism into existing CSP operators, eliminating the need for a mathematical proof. The results are thus based on empirical research in a field where mathematical proof is better known to be used. If the step laws of optional parallelism could be implemented in an existing CSP model-checker, a two-way trace refinement relationship between $OptPar \left(\uparrow \right)_X$ and $OptPar^T \left(\tilde{\parallel} \right)_X$ could be investigated. The results presented in this dissertation only confirms a one-way trace refinement relation between optional parallelism and its translation.

The complexity of a system definition is increased with the addition of the channel modelling artefacts used in $OptPar^T$, especially because of the use of the interleaving operator. CSP model-checkers such as ProCSP struggle to perform model-checking and deadlock analysis of large system definitions, which limits the applicability to test real-world WSN specifications where hundreds of nodes could form a WSN. This therefore solicits the need for optional parallelism to be implemented in CSP model-checkers so that it can be optimised for formal analysis.

It is concluded that optional parallelism opens new avenues to WSN modelling in CSP with the introduction to model broadcasting with pragmatic ease.

Future work regarding the study of optional parallelism includes the following:

- **Formal Proof** - The step laws of the optional parallelism given in [17] need to be formally verified with mathematics. If this could be done, its incorporation into model-checkers will be much easier and more widely accepted. This dissertation focussed more on the operational semantics of optional parallelism in the traces domain, but further research is required to develop the denotational and axiomatic semantics which are needed for this operator to become part of the classical CSP set of operators.
- **Other domains** - Optional parallelism should be investigated in other models of CSP such as the failures/divergences and the stable failures model. Initial work on failures and divergences were briefly given in [17] and could serve as a starting point for future development of optional parallelism in other CSP domains.
- **Termination** - Various approaches to termination of CSP has been attempted. Some of these approaches were mentioned in §2.2.5.5 and termination of optional parallelism was only briefly described in [17]. It would be useful to detail the termination semantics of optional parallelism together with the work mentioned in §2.2.5.5 to clarify possible unnatural process definitions where an event can occur after the termination event \checkmark . This is especially useful as optional parallelism incorporates semantics of interleaving which *does* allow for unnatural process definitions in terms of termination [77].
- **Model-Checkers** - The step laws of optional parallelism could be implemented into existing CSP model-checkers. If it is implemented in Prolog, it can be included into ProCSP. The work of this dissertation proved that optional parallelism can be used to model systems where partial synchronisation is needed. Tool support will make the optional parallel operator a better known and more frequently used operator of the CSP language.
- **Stochastic CSP** - The use of probabilities can be added to the channel artefacts so that the probability of communication or node failures can be modelled with CSP system definitions. Stochastic CSP is yet to be included in standard CSP model-checkers, but could prove useful if it could be combined with optional parallelism.

REFERENCES

- [1] T. Strauss, D. G. Kourie, and B. W. Watson, “A concurrent specification of an incremental DFA minimisation algorithm,” in *Proceedings of the Prague Stringology Conference 2008*, J. Holub and J. Ždárek, Eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 218–226.
- [2] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice-Hall (Pearson), 2005.
- [3] A. Intana, M. Poppleton, and G. Merrett, “Adding value to WSN simulation through formal modelling and analysis,” in *Software Engineering for Sensor Network Applications (SESENA), 2013 4th International Workshop on*, May 2013, pp. 24–29.
- [4] A. Testa, A. Coronato, M. Cinque, and J. Augusto, “Static Verification of Wireless Sensor Networks with Formal Methods,” in *Signal Image Technology and Internet Based Systems (SITIS), 2012 Eighth International Conference on*, Nov 2012, pp. 587–594.
- [5] M. Zheng, J. Sun, Y. Liu, J. Dong, and Y. Gu, “Towards a Model Checker for NesC and Wireless Sensor Networks,” in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science, S. Qin and Z. Qiu, Eds. Springer Berlin Heidelberg, 2011, vol. 6991, pp. 372–387.
- [6] S. Nanz and C. Hankin, “A framework for security analysis of mobile wireless networks,” *Theoretical Computer Science*, vol. 367, no. 1-2, pp. 203 – 227, 2006, automated Reasoning for Security Protocol Analysis Automated Reasoning for Security Protocol Analysis.
- [7] J. Godskesen, “A Calculus for Mobile Ad Hoc Networks,” in *Coordination Models and*

References

- Languages*, ser. Lecture Notes in Computer Science, A. Murphy and J. Vitek, Eds. Springer Berlin Heidelberg, 2007, vol. 4467, pp. 132–150.
- [8] A. Singh, C. Ramakrishnan, and S. Smolka, “A Process Calculus for mobile ad hoc networks,” in *Coordination Models and Languages*, ser. Lecture Notes in Computer Science, D. Lea and G. Zavattaro, Eds. Springer Berlin Heidelberg, 2008, vol. 5052, pp. 296–314.
- [9] M. Merro, “An observational theory for mobile ad hoc networks (full version),” *Information and Computation*, vol. 207, no. 2, pp. 194 – 208, 2009, special issue on Structural Operational Semantics (SOS).
- [10] M. Merro, F. Ballardin, and E. Sibilio, “A timed calculus for wireless systems,” *Theoretical Computer Science*, vol. 412, no. 47, pp. 6585 – 6611, 2011.
- [11] C. A. R. Hoare, *Communicating sequential processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.
- [12] M. G. Hinchey, J. L. Rash, C. A. Rouff, and D. Gracanin, “Achieving dependability in sensor networks through automated requirements-based programming,” *Computer Communications*, vol. 29, no. 2, pp. 246 – 256, 2006.
- [13] S. J. Creese and A. W. Roscoe, “Verifying an infinite family of inductions simultaneously using data independence and FDR,” in *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, ser. FORTE XII / PSTV XIX '99. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 1999, pp. 437–452.
- [14] I. Zakiuddin, M. Goldsmith, P. Whittaker, and P. Gardiner, “A methodology for model-checking ad-hoc networks,” in *Proceedings of the 10th international conference on Model checking software*, ser. SPIN'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 181–196.
- [15] S. Liu, X. Wu, Q. Li, H. Zhu, and Q. Wang, “Formal Approaches to Wireless Sensor Networks,” in *Secure Software Integration Reliability Improvement Companion (SSIRI-*

References

- C), *2011 5th International Conference on*, June 2011, pp. 11–18.
- [16] Y. Isobe and M. Roggenbach, “A generic theorem prover of CSP refinement,” in *In TACAS 2005, LNCS 3440*. Springer, 2005, pp. 108–123.
- [17] M. Roggenbach, S. Gruner, D. Kourie, T. Strauss, and B. Watson, “A New CSP Operator for Optional Parallelism,” in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 2, dec. 2008, pp. 788–791.
- [18] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [19] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless sensor networks: A survey,” *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.
- [20] A.-S. Pathan and C. Hong, “A Secure Energy-Efficient Routing Protocol for WSN,” in *Parallel and Distributed Processing and Applications*, ser. Lecture Notes in Computer Science, I. Stojmenovic, R. Thulasiram, L. Yang, W. Jia, M. Guo, and R. de Mello, Eds. Springer Berlin Heidelberg, 2007, vol. 4742, pp. 407–418.
- [21] E. Shih, S.-H. Cho, N. Ickes, R. Min, A. Sinha, A. Wang, and A. Chandrakasan, “Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks,” in *Proceedings of the 7th annual international conference on Mobile computing and networking*, ser. MobiCom ’01. New York, NY, USA: ACM, 2001, pp. 272–287.
- [22] D. Baghyalakshmi, J. Ebenezer, and S. A. V. SatyaMurthy, “Low latency and energy efficient routing protocols for wireless sensor networks,” in *Wireless Communication and Sensor Computing, 2010. ICWCSC 2010. International Conference on*, Jan 2010, pp. 1–6.
- [23] D. Singh and R. Goudar, “Energy efficient clearance routing in WSN,” *International Journal of System Assurance Engineering and Management*, pp. 1–26, 2014.
- [24] J. Liu, J. Li, X. Niu, X. Cui, and Y. Sun, “GreenOCR: An Energy-Efficient Optimal

References

- Clustering Routing Protocol,” *The Computer Journal*, 2014.
- [25] K. Mikhaylov and J. Tervonen, “Optimization of microcontroller hardware parameters for Wireless Sensor Network node power consumption and lifetime improvement,” in *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on*, Oct 2010, pp. 1150–1156.
- [26] C. Torres and P. Glosekotter, “Reliable and energy optimized WSN design for a train application,” *Journal of Systems Architecture*, vol. 57, no. 10, pp. 896 – 904, 2011, emerging Applications of Embedded Systems Research.
- [27] A. Castagnetti, A. Pegatoquet, C. Belleudy, and M. Auguin, “A framework for modeling and simulating energy harvesting WSN nodes with efficient power management policies,” *EURASIP Journal on Embedded Systems*, vol. 2012, no. 1, 2012.
- [28] P. De Mil, T. Allemeersch, I. Moerman, P. Demeester, and W. De Kimpe, “A Scalable Low-Power WSN Solution for Large-Scale Building Automation,” in *Communications, 2008. ICC '08. IEEE International Conference on*, May 2008, pp. 3130–3135.
- [29] E. Cayirci, “Wireless Sensor and Actuator Network Applications and Challenges,” in *Autonomous Sensor Networks*, ser. Springer Series on Chemical Sensors and Biosensors, D. Filippini, Ed. Springer Berlin Heidelberg, 2013, vol. 13, pp. 1–15.
- [30] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, “An analysis of a large scale habitat monitoring application,” in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 214–226.
- [31] J. L. Hill, “System architecture for wireless sensor networks,” Ph.D. dissertation, University of California, Berkeley, 2003.
- [32] D. Guyomar and M. Lallart, “Recent Progress in Piezoelectric Conversion and Energy Harvesting Using Nonlinear Electronic Interfaces and Issues in Small Scale Implementation,” *Micromachines*, vol. 2, no. 2, pp. 274–294, 2011.
- [33] S. Roundy, P. K. Wright, and J. Rabaey, “A study of low level vibrations as a power

References

- source for wireless sensor nodes,” *Comput. Commun.*, vol. 26, no. 11, pp. 1131–1144, Jul. 2003.
- [34] S. Roundy, D. Steingart, L. Frechette, P. Wright, and J. Rabaey, “Power Sources for Wireless Sensor Networks,” in *Wireless Sensor Networks*, ser. Lecture Notes in Computer Science, H. Karl, A. Wolisz, and A. Willig, Eds. Springer Berlin Heidelberg, 2004, vol. 2920, pp. 1–17.
- [35] R. Mulligan and H. M. Ammari, “Coverage in Wireless Sensor Networks: A Survey,” *Network Protocols and Algorithms*, vol. 2, no. 2, pp. 27–53, Apr. 2010.
- [36] B. Wang, “Coverage problems in sensor networks: A survey,” *ACM Comput. Surv.*, vol. 43, no. 4, pp. 32:1–32:53, Oct. 2011.
- [37] I. Chatzigiannakis, G. Mylonas, and S. Nikolettseas, “Modeling and evaluation of the effect of obstacles on the performance of wireless sensor networks,” in *Simulation Symposium, 2006. 39th Annual*, april 2006, p. 11 pp.
- [38] G. Hoblos, M. Staroswiecki, and A. Aitouche, “Optimal design of fault tolerant sensor networks,” in *Control Applications, 2000. Proceedings of the 2000 IEEE International Conference on*, 2000, pp. 467 –472.
- [39] T. Antoine-Santoni, J.-F. Santucci, E. De Gentili, X. Silvani, and F. Morandini, “Performance of a Protected Wireless Sensor Network in a Fire. Analysis of Fire Spread and Data Transmission,” *Sensors*, vol. 9, no. 8, pp. 5878–5893, 2009.
- [40] C. Clapham and J. Nicholson, *The Concise Oxford Dictionary of Mathematics, 4th Edition*. Oxford University Press, 2009.
- [41] S. Schwartzman, *The Words of Mathematics: An Etymological Dictionary of Mathematical Terms Used in English*, ser. MAA spectrum. Mathematical Association of America, 1994.
- [42] W. Fokkink, *Introduction to Process Algebra*, 1st ed. Springer Publishing Company, Incorporated, 2010.

References

- [43] W. Fokkink, J. F. Groote, and M. Reners, “Process Algebra Needs Proof Methodology (Columns: Concurrency).” *Bulletin of the EATCS*, vol. 82, pp. 109–125, 2004.
- [44] J. C. M. Baeten, “A brief history of process algebra,” *Theor. Comput. Sci.*, vol. 335, no. 2-3, pp. 131–146, May 2005.
- [45] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.
- [46] J. C. M. Baeten and W. P. Weijland, *Process Algebra*. New York, NY, USA: Cambridge University Press, 1990.
- [47] S. Schneider and R. Delicata, “Verifying security protocols: An application of CSP,” in *Proceedings of the 2004 international conference on Communicating Sequential Processes: the First 25 Years*, ser. CSP’04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 243–263.
- [48] A. Simpson, “The application of formal methods to the development of an ATP (automatic train protection) system,” in *Communication Networks in Transportation, IEE Colloquium on*, Jan 1995, pp. 5/1–5/4.
- [49] —, “Model Checking for Interlocking Safety,” in *In Proceedings of the Second FMERail Seminar*, 1998, pp. 15–16.
- [50] W. Su, F. Yang, X. Wu, J. Guo, and H. Zhu, “Formal Approaches to Mode Conversion and Positioning for Vehicle System,” in *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, July 2011, pp. 416–421.
- [51] P. Welch, G. Hilderink, A. Bakkers, and G. Stiles, “Safe and Verifiable Design of Concurrent Java Programs,” *International Journal of Computers and Application*, vol. 23, no. 3, pp. 159–165, 2001.
- [52] A. I. McInnes, “Using CSP to Model and Analyze TinyOS Applications,” in *Proceedings of the 2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ser. ECBS ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 79–88.

References

- [53] A. W. Roscoe and N. Dathi, “The pursuit of deadlock freedom,” *Inf. Comput.*, vol. 75, no. 3, pp. 289–327, Dec. 1987.
- [54] S. Gruner and T. J. Steyn, “Deadlock-freeness of hexagonal systolic arrays,” *Inf. Process. Lett.*, vol. 110, no. 14-15, pp. 539–543, Jul. 2010.
- [55] H. T. Kung and C. E. Leiserson, “Systolic arrays (for VLSI),” in *Sparse matrix proceedings, 1978*, G. W. Duff, I. S. and Stewart, Ed. Philadelphia, USA: Society for Industrial and Applied Mathematics, Dec. 1978, pp. 256–282.
- [56] H. R. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*. New York, NY, USA: John Wiley & Sons, Inc., 1992.
- [57] S. Ripon, “Extending and Relating Semantic Models of Compensating CSP,” Ph.D. dissertation, University of Southampton, August 2008.
- [58] G. D. Plotkin, “A structural approach to operational semantics,” *The Journal of Logic and Algebraic Programming*, vol. 60-61, pp. 17–139, Dec. 2004.
- [59] P. D. Mosses, “Formal semantics of programming languages: An overview,” *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 41 – 73, 2006, proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004) Foundations of Visual Modelling Techniques 2004.
- [60] G. Kahn, “Natural Semantics,” in *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, ser. STACS ’87. London, UK, UK: Springer-Verlag, 1987, pp. 22–39.
- [61] D. Scott and C. Strachey, “Toward A Mathematical Semantics for Computer Languages,” in *Proceedings of the Symposium on Computers and Automata*, J. Fox, Ed., vol. XXI. Brooklyn, N.Y.: Polytechnic Press, Apr. 1971, pp. 19–46.
- [62] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.
- [63] G. M. Reed, “A uniform mathematical theory for real-time distributed computing,”

References

- Ph.D. dissertation, University of Oxford, 1988, aAID-85967.
- [64] A. Lawrence, “Extending CSP: Denotational Semantics,” *Software, IEE Proceedings* -, vol. 150, no. 2, pp. 51 – 60, April 2003.
- [65] A. W. Roscoe, “On The Expressiveness of CSP,” February 2011.
- [66] M. O. Larsen, “Exception Handling in Communicating Sequential Processes,” Master’s thesis, University of Copenhagen, Copenhagen, Denmark, August 2012.
- [67] G. M. Reed and A. W. Roscoe, “A Timed Model for Communicating Sequential Processes,” in *ICALP*, ser. Lecture Notes in Computer Science, L. Kott, Ed., vol. 226. Springer, 1986, pp. 314–323.
- [68] —, “Metric Spaces as Models for Real-Time Concurrency,” in *Proceedings of the 3rd Workshop on Mathematical Foundations of Programming Language Semantics*. London, UK, UK: Springer-Verlag, 1988, pp. 331–343.
- [69] J. Davies and S. Schneider, “An introduction to timed CSP,” OUCL, Tech. Rep. PRG75, August 1989.
- [70] —, “A brief history of timed CSP,” *Theor. Comput. Sci.*, vol. 138, no. 2, pp. 243–271, 1995.
- [71] S. Schneider, “An Operational Semantics for Timed CSP,” *Information and Computation*, vol. 116, no. 2, pp. 193 – 213, 1995.
- [72] J. Ouaknine and S. Schneider, “Timed CSP: A Retrospective,” *Electronic Notes in Theoretical Computer Science*, vol. 162, pp. 273–276, Sep. 2006.
- [73] K. Prasad, “A calculus of broadcasting systems,” *Science of Computer Programming*, vol. 25, no. 2 - 3, pp. 285 – 327, 1995, selected Papers of ESOP’94, the 5th European Symposium on Programming.
- [74] M. Butler, T. Hoare, and C. Ferreira, “A Trace Semantics for Long-Running Transactions,” in *Communicating Sequential Processes. The First 25 Years*, ser. Lecture Notes in Computer Science, A. Abdallah, C. Jones, and J. Sanders, Eds. Springer Berlin

References

- Heidelberg, 2005, vol. 3525, pp. 133–150.
- [75] M. Butler and S. Ripon, “Executable Semantics for Compensating CSP,” in *Formal Techniques for Computer Systems and Business Processes*, ser. Lecture Notes in Computer Science, M. Bravetti, L. Kloul, and G. Zavattaro, Eds. Springer Berlin Heidelberg, 2005, vol. 3670, pp. 243–256.
- [76] K. Wan, H. K. Kapoor, S. Das, B. Raju, T. Krilavicius, and K. L. Man, “Modelling and Verification of Compensating Transactions using the Spin Tool,” in *In Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. II, Hong Kong, Jul 2012.
- [77] P. Howells and M. d’Inverno, “A CSP model with flexible parallel termination semantics,” *Formal Asp. Comput.*, vol. 21, no. 5, pp. 421–449, 2009.
- [78] H. Tej and B. Wolff, “A Corrected Failure-Divergence Model for CSP in Isabelle/HOL,” in *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, ser. LNCS 1313, J. Fitzgerald, C. Jones, and P. Lucas, Eds. Springer Verlag, 1997, pp. 318–337.
- [79] C. Lüth, H. Shi, and H. Tej, “Formal Development of Processes by Model-Checking and Theorem Proving with FDR and HOL-CSP,” *Inf. Comput.*, vol. 75, no. 3, pp. 289–327, Dec. 1987.
- [80] Y. Isobe and M. Roggenbach, “A Generic Theorem Prover of CSP Refinement,” in *TACAS*, ser. Lecture Notes in Computer Science, N. Halbwegs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 108–123.
- [81] L. C. Paulson, *Isabelle: a Generic Theorem Prover*, ser. Lecture Notes in Computer Science. Springer – Berlin, 1994, no. 828.
- [82] Y. Isobe, M. Roggenbach, and S. Gruner, “Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays,” in *FOSE 2005*, ser. Japanese Lecture Notes Series 31. Kindai-kagaku-sha, 2005.
- [83] F. S. E. Limited. (2009, Sep.) Failures-Divergence Refinement: FDR2. [Online].

References

- Available: <http://www.fsel.com>
- [84] A. Valmari, “The State Explosion Problem,” in *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*. London, UK: Springer-Verlag, 1998, pp. 429–528.
- [85] M. Goldsmith and J. Martin, “Parallelization of FDR,” *Workshop on Parallel and Distributed Model Checking, affiliated to CONCUR*, 2002.
- [86] A. B. A. R. Thomas Gibson-Robinson, Philip Armstrong, “FDR3 — A Modern Refinement Checker for CSP,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. ÅbrahÅm and K. Havelund, Eds., vol. 8413, 2014, pp. 187–201.
- [87] —, *Failures Divergences Refinement (FDR) Version 3*, 2013. [Online]. Available: <https://www.cs.ox.ac.uk/projects/fdr/>
- [88] M. Leuschel and M. Butler, “ProB: A Model Checker for B,” in *FME 2003: Formal Methods*, ser. LNCS 2805, K. Araki, S. Gnesi, and D. Mandrioli, Eds. Springer-Verlag, 2003, pp. 855–874.
- [89] R. Perlman, *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols (2nd Edition)*, 2nd ed. Addison-Wesley Professional, Sep. 1999.
- [90] S. Creese, “Data Independent Induction: CSP Model Checking of Arbitrary Sized Networks,” Ph.D. dissertation, University of Oxford, Computing Laboratory, 2001.
- [91] S. Curtis, J. Mica, J. Nuth, G. Marr, M. Rilee, and M. Bhat, “ANTS (Autonomous Nano-Technology Swarm): An artificial intelligence approach to asteroid belt resource exploration,” in *International Astronautical Federation, 51th Congress*, 2000.
- [92] C. A. Rouff, A. Vanderbilt, W. Truszkowski, J. L. Rash, and M. G. Hinchey, “Formal Methods for Autonomic and Swarm-based Systems,” in *ISoLA (Preliminary proceedings)*, ser. Technical Report, T. Margaria, B. Steffen, A. Philippou, and M. Reitenspieß, Eds., vol. TR-2004-6. Department of Computer Science, University of Cyprus, 2004, pp. 100–102.

References

- [93] J. S. Dong, J. Sun, J. Sun, K. Taguchi, and X. Zhang, “Specifying and Verifying Sensor Networks: An Experiment of Formal Methods,” in *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ser. ICFEM ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 318–337.
- [94] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe, “Timed CSP: Theory and Practice,” in *Proceedings of the Real-Time: Theory in Practice, REX Workshop*. London, UK, UK: Springer-Verlag, 1992, pp. 640–675.
- [95] B. Mahony and J. S. Dong, “Timed Communicating Object Z,” *Software Engineering, IEEE Transactions on*, vol. 26, no. 2, pp. 150–177, 2000.
- [96] S. Jasko and G. Simon, “CSP-Based Sensor Network Architecture for Reconfigurable Measurement Systems,” *IEEE Transactions on Instrumentation and Measurement*, vol. 60, pp. 2104–2117, 2011.
- [97] K. G. Larsen, P. Pettersson, and W. Yi, “Uppaal in a Nutshell,” *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, Oct. 1997.
- [98] M. Botts and A. Robin, “OpenGIS Sensor Model Language SensorML Implementation Specification,” 2007.
- [99] S. Jasko and G. Simon, “Reconfigurable sensor network architecture for distributed measurement systems,” in *Instrumentation and Measurement Technology Conference (I2MTC), 2010 IEEE*, may 2010, pp. 198 –203.
- [100] J. M. R. Martin and P. H. Welch, “A Design Strategy for Deadlock-Free Concurrent Systems,” *Transputer Communications*, vol. 3, no. 4, pp. 215–232, 1997.
- [101] A. Basu, L. Mounier, M. Poulhies, J. Pulou, and J. Sifakis, “Using BIP for Modeling and Verification of Networked Systems - A Case Study on TinyOS-based Networks,” in *IEEE NCA ’07*, Cambridge, USA, July 2007, pp. 257–260.
- [102] A. Basu, M. Bozga, and J. Sifakis, “Modeling Heterogeneous Real-Time Components in BIP,” in *In 4 th IEEE International Conference on Software Engineering and Formal Methods (SEFM06)*, 2006, pp. 3–12.

References

- [103] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesC Language: A Holistic Approach to Networked Embedded Systems,” *SIGPLAN Not.*, vol. 38, no. 5, pp. 1–11, May 2003.
- [104] S. Bliudze and J. Sifakis, “The Algebra of Connectors: Structuring Interaction in BIP,” in *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, ser. EMSOFT '07. New York, NY, USA: ACM, 2007, pp. 11–20.
- [105] M. Shanahan, “The Event Calculus Explained,” in *Artificial Intelligence Today*, ser. Lecture Notes in Computer Science, M. Wooldridge and M. Veloso, Eds. Springer Berlin Heidelberg, 1999, vol. 1600, pp. 409–430.
- [106] P. Boonma and J. Suzuki, “Model-driven performance engineering for wireless sensor networks with feature modeling and event calculus,” in *Proceedings of the 3rd Workshop on Biologically Inspired Algorithms for Distributed Systems*, ser. BADS '11. New York, NY, USA: ACM, 2011, pp. 17–24.
- [107] Y. Choe and M. Lee, “ δ -Calculus: Process Algebra to model Secure Movements of Distributed Mobile Processes in Real-Time Business Applications.” in *Proceedings 23rd ECIS'2015: European Conference on Information Systems*, ser. ECIS'2015, vol. 8413, May 2015, pp. 187–201.
- [108] E. G. Coffman, M. Elphick, and A. Shoshani, “System Deadlocks,” *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, Jun. 1971.
- [109] R. Dechter, “Decomposing an N-ary relation into a tree of binary relations,” in *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, ser. PODS '87. New York, NY, USA: ACM, 1987, pp. 185–189.
- [110] R. Sedgewick and K. Wayne, *Algorithms, 4th Edition*. Addison-Wesley, 2011.
- [111] D. Leijen, W. Schulte, and S. Burckhardt, “The design of a task parallel library,” *SIGPLAN Not.*, vol. 44, no. 10, pp. 227–242, Oct. 2009.
- [112] D. E. Knuth, *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

References

- [113] Q. Mamun, “A Qualitative Comparison of Different Logical Topologies for Wireless Sensor Networks,” *Sensors*, vol. 12, no. 11, pp. 14 887–14 913, 2012.
- [114] T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*, 1st ed. Springer Publishing Company, Incorporated, 2008.
- [115] A. Varga and R. Hornig, “An overview of the OMNeT++ simulation environment,” in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, ser. Simutools '08. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 60:1–60:10.

APPENDIX A

TRACE RESULTS

Below is the trace results of $traces(P \parallel Q \parallel R)$ described in §5.2.4.1.1.

{ <>, <mr>, <mp>, <mq>
 <mr, DL>, <mq, mr>, <mp, mr>, <mr, mp>, <mp, mq>, <mr, mq>, <mq, mp>,
 <mp, mr, DL>, <mr, mp, DL>, <mp, mq, mr>, <mq, mr, DL>, <mr, mq, DL>,
 <mq, mp, mr>, <mp, mr, mq>, <mr, mp, mq>, <mq, mr, mp>, <mr, mq, mp>,
 <mp, mq, mr, DL>, <mp, mr, mq, DL>, <mr, mp, mq, DL>, <mp, mq, mr, s>,
 <mp, mr, mq, s>, <mr, mp, mq, s>, <mq, mp, mr, DL>, <mq, mr, mp, DL>,
 <mr, mq, mp, DL>, <mq, mp, mr, s>, <mq, mr, mp, s>, <mr, mq, mp, s>,
 <mp, mq, mr, s, wr>, <mp, mr, mq, s, wr>, <mr, mp, mq, s, wr>,
 <mp, mq, mr, s, wp>, <mp, mr, mq, s, wp>, <mr, mp, mq, s, wp>,
 <mp, mq, mr, s, wq>, <mr, mp, mq, s, wq>, <mq, mp, mr, s, wr>,
 <mq, mr, mp, s, wr>, <mr, mq, mp, s, wr>, <mq, mp, mr, s, wp>,
 <mq, mr, mp, s, wp>, <mr, mq, mp, s, wp>, <mq, mp, mr, s, wq>,
 <mq, mr, mp, s, wq>, <mr, mq, mp, s, wq>, <mp, mr, mq, s, wq>,
 <mp, mq, mr, s, wp, wr>, <mp, mq, mr, s, wr, wp>, <mp, mr, mq, s, wp, wr>,
 <mp, mr, mq, s, wr, wp>, <mr, mp, mq, s, wp, wr>, <mr, mp, mq, s, wr, wp>,
 <mp, mq, mr, s, wp, wq>, <mp, mr, mq, s, wp, wq>, <mr, mp, mq, s, wp, wq>,
 <mp, mq, mr, s, wq, wr>, <mp, mq, mr, s, wr, wq>, <mp, mr, mq, s, wq, wr>,
 <mp, mr, mq, s, wr, wq>, <mr, mp, mq, s, wq, wr>, <mr, mp, mq, s, wr, wq>,
 <mp, mq, mr, s, wq, wp>, <mp, mr, mq, s, wq, wp>, <mr, mp, mq, s, wq, wp>,
 <mq, mp, mr, s, wp, wr>, <mq, mp, mr, s, wr, wp>, <mq, mr, mp, s, wp, wr>,
 <mq, mr, mp, s, wr, wp>, <mr, mq, mp, s, wp, wr>, <mr, mq, mp, s, wr, wp>,

$\langle mq, mp, mr, s, wp, wq \rangle, \langle mq, mr, mp, s, wp, wq \rangle, \langle mr, mq, mp, s, wp, wq \rangle,$
 $\langle mq, mp, mr, s, wq, wr \rangle, \langle mq, mp, mr, s, wr, wq \rangle, \langle mq, mr, mp, s, wq, wr \rangle,$
 $\langle mq, mr, mp, s, wr, wq \rangle, \langle mr, mq, mp, s, wq, wr \rangle, \langle mr, mq, mp, s, wr, wq \rangle,$
 $\langle mq, mp, mr, s, wq, wp \rangle, \langle mq, mr, mp, s, wq, wp \rangle, \langle mr, mq, mp, s, wq, wp \rangle,$
 $\langle mp, mq, mr, s, wp, wq, wr \rangle, \langle mp, mq, mr, s, wp, wr, wq \rangle, \langle mp, mq, mr, s, wr, wp, wq \rangle,$
 $\langle mp, mr, mq, s, wp, wq, wr \rangle, \langle mp, mr, mq, s, wp, wr, wq \rangle, \langle mp, mr, mq, s, wr, wp, wq \rangle,$
 $\langle mr, mp, mq, s, wp, wq, wr \rangle, \langle mr, mp, mq, s, wp, wr, wq \rangle, \langle mr, mp, mq, s, wr, wp, wq \rangle,$
 $\langle mp, mq, mr, s, wq, wp, wr \rangle, \langle mp, mq, mr, s, wq, wr, wp \rangle, \langle mp, mq, mr, s, wr, wq, wp \rangle,$
 $\langle mp, mr, mq, s, wq, wp, wr \rangle, \langle mp, mr, mq, s, wq, wr, wp \rangle, \langle mp, mr, mq, s, wr, wq, wp \rangle,$
 $\langle mr, mp, mq, s, wq, wp, wr \rangle, \langle mr, mp, mq, s, wq, wr, wp \rangle, \langle mr, mp, mq, s, wr, wq, wp \rangle,$
 $\langle mq, mp, mr, s, wp, wq, wr \rangle, \langle mq, mp, mr, s, wp, wr, wq \rangle, \langle mq, mp, mr, s, wr, wp, wq \rangle,$
 $\langle mq, mr, mp, s, wp, wq, wr \rangle, \langle mq, mr, mp, s, wp, wr, wq \rangle, \langle mq, mr, mp, s, wr, wp, wq \rangle,$
 $\langle mr, mq, mp, s, wp, wq, wr \rangle, \langle mr, mq, mp, s, wp, wr, wq \rangle, \langle mr, mq, mp, s, wr, wp, wq \rangle,$
 $\langle mq, mp, mr, s, wq, wp, wr \rangle, \langle mq, mp, mr, s, wq, wr, wp \rangle, \langle mq, mp, mr, s, wr, wq, wp \rangle,$
 $\langle mq, mr, mp, s, wq, wp, wr \rangle, \langle mq, mr, mp, s, wq, wr, wp \rangle, \langle mq, mr, mp, s, wr, wq, wp \rangle,$
 $\langle mr, mq, mp, s, wq, wp, wr \rangle, \langle mr, mq, mp, s, wq, wr, wp \rangle, \langle mr, mq, mp, s, wr, wq, wp \rangle$

APPENDIX B

TOPOLOGY SCENARIO MODELS

This appendix gives the CSP_M models of the topology scenarios presented in Chapter 6. The models were generated by OpTrace, with the given adjacency list definitions and the mode of synchronisation, i.e. broadcasting, half-duplex or simplex.

B.1 FLAT TOPOLOGY

B.1.1 Point-to-point

B.1.1.1 Broadcasting

```
-----  
-- Auto generated CSPM model by OpTrace  
-- Broadcasting  
-- Input:  
-- (P,Q)  
-----
```

```
channel Ap,A,Aq
```

```
aP = { | Ap | }
```

```
NodeP = ( Ap -> NodeP )
```

```
aQ = { | Aq | }
```

```
NodeQ = ( Aq -> NodeQ )
```

```
aCPA = { | Ap, A | }
```

```

ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCQA = { | Aq, A | }

ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

SYSTEM = ( NodeP ||| NodeQ ) [ | {Ap, Aq} | ] (ChanP_A [ { |Ap,A| } | | { |Aq,A| } ] ↔
  ChanQ_A)
MAIN = SYSTEM
  
```

Listing B.1: Generated CSP_M model of Listing 6.1.

B.1.1.2 Bidirectional - Half-duplex

```

-----
-- Auto generated CSPM model by OpTrace
-- HalfDuplex
-- Input:
-- {P,Q}
-----

channel Ap,A,Aq

aP = { | Ap | }
NodeP = ( Ap -> NodeP )

aQ = { | Aq | }
NodeQ = ( Aq -> NodeQ )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A)) [ ] (A -> (Ap -> ChanP_A [] ChanP_A))

aCQA = { | Aq, A | }
ChanQ_A = (Aq -> (A -> ChanQ_A [] ChanQ_A)) [ ] (A -> (Aq -> ChanQ_A [] ChanQ_A))

SYSTEM = ( NodeP ||| NodeQ ) [ | {Ap, Aq} | ] (ChanP_A [ { |Ap,A| } | | { |Aq,A| } ] ↔
  ChanQ_A)
MAIN = SYSTEM
  
```


Listing B.2: Generated CSP_M model of Listing 6.2.

B.1.1.3 Bidirectional - Simplex

```

-----
-- Auto generated CSPM model by OpTrace
-- Simplex
-- Input:
-- (P,Q)
-- (Q,P)
-----

channel Ap, A, Bp, B, Aq, Bq

aP = { | Ap, Bp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP )

aQ = { | Bq, Aq | }
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (B -> (Bp -> ChanP_B [] ChanP_B))

aCQA = { | Aq, A | }
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQB = { | Bq, B | }
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))

SYSTEM = ( NodeP ||| NodeQ ) [] { Ap, Bp, Aq, Bq } [] ((ChanP_A [ { | Ap, A | } || { | Bp, B ←
    | } ] ChanP_B) [ { | Ap, A, Bp, B | } || { | Aq, A | } ] ChanQ_A) [ { | Ap, A, Bp, B, Aq | } || { | Bq, ←
    B | } ] ChanQ_B)
MAIN = SYSTEM

```

Listing B.3: Generated CSP_M model of Listing 6.3.

B.1.2 Fully Connected Mesh
B.1.2.1 Broadcasting

```

-----
-- Auto generated CSPM model by OpTrace
-- Broadcasting
-- Input:
-- (P,Q,R,S)
-- (Q,P,R,S)
-- (R,P,Q,S)
-- (S,P,Q,R)
-----

channel Ap,A,Bp,B,Cp,C,Dp,D,Aq,Bq,Cq,Dq,Ar,Br,Cr,Dr,As,Bs,Cs,Ds

aP = { | Ap,Bp,Cp,Dp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP ) [] ( Cp -> NodeP ) [] ( Dp -> NodeP )

aQ = { | Bq,Aq,Cq,Dq | }
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ ) [] ( Cq -> NodeQ ) [] ( Dq -> NodeQ )

aR = { | Cr,Ar,Br,Dr | }
NodeR = ( Ar -> NodeR ) [] ( Br -> NodeR ) [] ( Cr -> NodeR ) [] ( Dr -> NodeR )

aS = { | Ds,As,Bs,Cs | }
NodeS = ( As -> NodeS ) [] ( Bs -> NodeS ) [] ( Cs -> NodeS ) [] ( Ds -> NodeS )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (B -> (Bp -> ChanP_B [] ChanP_B))
  
```

```

aCPC = { | Cp, C | }
ChanP_C = (C -> (Cp -> ChanP_C [] ChanP_C))

aCPD = { | Dp, D | }
ChanP_D = (D -> (Dp -> ChanP_D [] ChanP_D))

aCQA = { | Aq, A | }
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQB = { | Bq, B | }
ChanQ_B = (B -> (Bq -> ChanQ_B [] ChanQ_B))

aCQC = { | Cq, C | }
ChanQ_C = (C -> (Cq -> ChanQ_C [] ChanQ_C))

aCQD = { | Dq, D | }
ChanQ_D = (D -> (Dq -> ChanQ_D [] ChanQ_D))

aCRA = { | Ar, A | }
ChanR_A = (A -> (Ar -> ChanR_A [] ChanR_A))

aCRB = { | Br, B | }
ChanR_B = (B -> (Br -> ChanR_B [] ChanR_B))

aCRC = { | Cr, C | }
ChanR_C = (C -> (Cr -> ChanR_C [] ChanR_C))

aCRD = { | Dr, D | }
ChanR_D = (D -> (Dr -> ChanR_D [] ChanR_D))

aCSA = { | As, A | }
ChanS_A = (A -> (As -> ChanS_A [] ChanS_A))

aCSB = { | Bs, B | }
ChanS_B = (B -> (Bs -> ChanS_B [] ChanS_B))

aCSC = { | Cs, C | }

```



```

NodeQ = ( Aq -> NodeQ )

aR = { | Ar | }
NodeR = ( Ar -> NodeR )

aS = { | As | }
NodeS = ( As -> NodeS )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A)) [] (A -> (Ap -> ChanP_A [] ChanP_A))

aCQA = { | Aq, A | }
ChanQ_A = (Aq -> (A -> ChanQ_A [] ChanQ_A)) [] (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCRA = { | Ar, A | }
ChanR_A = (Ar -> (A -> ChanR_A [] ChanR_A)) [] (A -> (Ar -> ChanR_A [] ChanR_A))

aCSA = { | As, A | }
ChanS_A = (As -> (A -> ChanS_A [] ChanS_A)) [] (A -> (As -> ChanS_A [] ChanS_A))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ) [] {Ap, Aq, Ar, As} [] (((ChanP_A [←
    { |Ap,A| } || { |Aq,A| } ] ChanQ_A) [ { |Ap,A,Aq| } || { |Ar,A| } ] ChanR_A) [ { |Ap,A,←
    Aq,Ar| } || { |As,A| } ] ChanS_A)

MAIN = SYSTEM
  
```

Listing B.5: Generated CSP_M model of Listing 6.5.

B.1.2.3 Bidirectional - Simplex

```

-----
-- Auto generated CSPM model by OpTrace
-- Simplex
-- Input:
-- (P,Q)
-- (P,R)
-- (P,S)
-- (Q,P)
-- (Q,R)
  
```

```

-- (Q,S)
-- (R,P)
-- (R,Q)
-- (R,S)
-- (S,P)
-- (S,Q)
-- (S,R)

-----

channel Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr, Ir, I, Lr, L, Cs, Fs, Is, Js, Ks, Ls

aP = { | Ap, Bp, Cp, Dp, Gp, Jp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP ) [] ( Cp -> NodeP ) [] ( Dp -> NodeP ) [] ( Gp ↔
-> NodeP ) [] ( Jp -> NodeP )

aQ = { | Dq, Eq, Fq, Aq, Hq, Kq | }
NodeQ = ( Aq -> NodeQ ) [] ( Dq -> NodeQ ) [] ( Eq -> NodeQ ) [] ( Fq -> NodeQ ) [] ( Hq ↔
-> NodeQ ) [] ( Kq -> NodeQ )

aR = { | Gr, Hr, Ir, Br, Er, Lr | }
NodeR = ( Br -> NodeR ) [] ( Er -> NodeR ) [] ( Gr -> NodeR ) [] ( Hr -> NodeR ) [] ( Ir ↔
-> NodeR ) [] ( Lr -> NodeR )

aS = { | Js, Ks, Ls, Cs, Fs, Is | }
NodeS = ( Cs -> NodeS ) [] ( Fs -> NodeS ) [] ( Is -> NodeS ) [] ( Js -> NodeS ) [] ( Ks ↔
-> NodeS ) [] ( Ls -> NodeS )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (Bp -> (B -> ChanP_B [] ChanP_B))

aCPC = { | Cp, C | }
ChanP_C = (Cp -> (C -> ChanP_C [] ChanP_C))

```

```

aCPD = { | Dp, D | }
ChanP_D = (D -> (Dp -> ChanP_D [] ChanP_D))

aCPG = { | Gp, G | }
ChanP_G = (G -> (Gp -> ChanP_G [] ChanP_G))

aCPJ = { | Jp, J | }
ChanP_J = (J -> (Jp -> ChanP_J [] ChanP_J))

aCQA = { | Aq, A | }
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQD = { | Dq, D | }
ChanQ_D = (Dq -> (D -> ChanQ_D [] ChanQ_D))

aCQE = { | Eq, E | }
ChanQ_E = (Eq -> (E -> ChanQ_E [] ChanQ_E))

aCQF = { | Fq, F | }
ChanQ_F = (Fq -> (F -> ChanQ_F [] ChanQ_F))

aCQH = { | Hq, H | }
ChanQ_H = (H -> (Hq -> ChanQ_H [] ChanQ_H))

aCQK = { | Kq, K | }
ChanQ_K = (K -> (Kq -> ChanQ_K [] ChanQ_K))

aCRB = { | Br, B | }
ChanR_B = (B -> (Br -> ChanR_B [] ChanR_B))

aCRE = { | Er, E | }
ChanR_E = (E -> (Er -> ChanR_E [] ChanR_E))

aCRG = { | Gr, G | }
ChanR_G = (Gr -> (G -> ChanR_G [] ChanR_G))

aCRH = { | Hr, H | }

```

```

ChanR_H = (Hr -> (H -> ChanR_H [] ChanR_H))

aCRI = { | Ir, I | }
ChanR_I = (Ir -> (I -> ChanR_I [] ChanR_I))

aCRL = { | Lr, L | }
ChanR_L = (L -> (Lr -> ChanR_L [] ChanR_L))

aCSC = { | Cs, C | }
ChanS_C = (C -> (Cs -> ChanS_C [] ChanS_C))

aCSF = { | Fs, F | }
ChanS_F = (F -> (Fs -> ChanS_F [] ChanS_F))

aCSI = { | Is, I | }
ChanS_I = (I -> (Is -> ChanS_I [] ChanS_I))

aCSJ = { | Js, J | }
ChanS_J = (J -> (J -> ChanS_J [] ChanS_J))

aCSK = { | Ks, K | }
ChanS_K = (K -> (K -> ChanS_K [] ChanS_K))

aCSL = { | Ls, L | }
ChanS_L = (L -> (L -> ChanS_L [] ChanS_L))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ) [ | {Ap, Bp, Cp, Dp, Gp, Jp, Aq, Dq, ←
    Eq, Fq, Hq, Kq, Br, Er, Gr, Hr, Ir, Lr, Cs, Fs, Is, Js, Ks, Ls} | | ←
    (((((((((((((((((((((((((ChanP_A [ { |Ap,A| } || { |Bp,B| } ] ChanP_B) [ { |Ap,A,Bp,B| } ←
    | | { |Cp,C| } ] ChanP_C) [ { |Ap,A,Bp,B,Cp,C| } || { |Dp,D| } ] ChanP_D) [ { |Ap,A,Bp, ←
    B,Cp,C,Dp,D| } || { |Gp,G| } ] ChanP_G) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Gp,G| } || { |Jp,J| } ←
    ] ChanP_J) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Gp,G,Jp,J| } || { |Aq,A| } ] ChanQ_A) [ { |Ap,A, ←
    Bp,B,Cp,C,Dp,D,Gp,G,Jp,J,Aq| } || { |Dq,D| } ] ChanQ_D) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Gp, ←
    G,Jp,J,Aq,Dq| } || { |Eq,E| } ] ChanQ_E) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Gp,G,Jp,J,Aq,Dq,Eq ←
    ,E| } || { |Fq,F| } ] ChanQ_F) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Gp,G,Jp,J,Aq,Dq,Eq,E,Fq,F| } ←
    | | { |Hq,H| } ] ChanQ_H) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Gp,G,Jp,J,Aq,Dq,Eq,E,Fq,F,Hq,H| } ←
    | | { |Kq,K| } ] ChanQ_K) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Gp,G,Jp,J,Aq,Dq,Eq,E,Fq,F,Hq,H,Kq ←

```



```

, K| } || { |Br, B| } ] ChanR_B) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br| } || { |Er, E| } ] ChanR_E) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er| } || { |Gr, G| } ] ChanR_G) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr| } || { |Hr, H| } ] ChanR_H) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr| } || { |Ir, I| } ] ChanR_I) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr, Ir, I| } || { |Lr, L| } ] ChanR_L) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr, Ir, I, Lr, L| } || { |Cs, C| } ] ChanS_C) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr, Ir, I, Lr, L, Cs| } || { |Fs, F| } ] ChanS_F) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr, Ir, I, Lr, L, Cs, Fs| } || { |Is, I| } ] ChanS_I) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr, Ir, I, Lr, L, Cs, Fs, Is| } || { |Js, J| } ] ChanS_J) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr, Ir, I, Lr, L, Cs, Fs, Is, Js| } || { |Ks, K| } ] ChanS_K) [ { |Ap, A, Bp, B, Cp, C, Dp, D, Gp, G, Jp, J, Aq, Dq, Eq, E, Fq, F, Hq, H, Kq, K, Br, Er, Gr, Hr, Ir, I, Lr, L, Cs, Fs, Is, Js, Ks| } || { |Ls, L| } ] ChanS_L)
MAIN = SYSTEM

```

Listing B.6: Generated CSP_M model of Listing 6.6.

B.2 CLUSTER TOPOLOGY

B.2.1 Star

B.2.1.1 Broadcasting

```

-----
-- Auto generated CSPM model by OpTrace
-- Broadcasting
-- Input:
-- (P, Q, R, S)
-- (Q, P)
-- (R, P)
-- (S, P)
-----

channel Ap, A, Bp, B, Cp, C, Dp, D, Aq, Bq, Ar, Cr, As, Ds

```

```

aP = { | Ap,Bp,Cp,Dp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP ) [] ( Cp -> NodeP ) [] ( Dp -> NodeP )

aQ = { | Bq,Aq | }
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ )

aR = { | Cr,Ar | }
NodeR = ( Ar -> NodeR ) [] ( Cr -> NodeR )

aS = { | Ds,As | }
NodeS = ( As -> NodeS ) [] ( Ds -> NodeS )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (B -> (Bp -> ChanP_B [] ChanP_B))

aCPC = { | Cp, C | }
ChanP_C = (C -> (Cp -> ChanP_C [] ChanP_C))

aCPD = { | Dp, D | }
ChanP_D = (D -> (Dp -> ChanP_D [] ChanP_D))

aCQA = { | Aq, A | }
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQB = { | Bq, B | }
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))

aCRA = { | Ar, A | }
ChanR_A = (A -> (Ar -> ChanR_A [] ChanR_A))

aCRC = { | Cr, C | }
ChanR_C = (Cr -> (C -> ChanR_C [] ChanR_C))

aCSA = { | As, A | }

```

```

ChanS_A = ( A -> (As -> ChanS_A [] ChanS_A) )

aCSD = { | Ds, D | }

ChanS_D = (Ds -> (D -> ChanS_D [] ChanS_D) )

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ) [ | {Ap, Bp, Cp, Dp, Aq, Bq, Ar, Cr, ←
  As, Ds} | ] ( ( ( ( ( ( ( ( ChanP_A [ { |Ap,A| } | | { |Bp,B| } ] ChanP_B) [ { |Ap,A,Bp,B| } ←
  | | { |Cp,C| } ] ChanP_C) [ { |Ap,A,Bp,B,Cp,C| } | | { |Dp,D| } ] ChanP_D) [ { |Ap,A,Bp,B, ←
  , Cp, C, Dp, D| } | | { |Aq,A| } ] ChanQ_A) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Aq| } | | { |Bq,B| } ] ←
  ChanQ_B) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Aq,Bq| } | | { |Ar,A| } ] ChanR_A) [ { |Ap,A,Bp,B,Cp, ←
  , C, Dp, D, Aq, Bq, Ar| } | | { |Cr,C| } ] ChanR_C) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Aq,Bq,Ar,Cr| } ←
  | | { |As,A| } ] ChanS_A) [ { |Ap,A,Bp,B,Cp,C,Dp,D,Aq,Bq,Ar,Cr,As| } | | { |Ds,D| } ] ←
  ChanS_D)

MAIN = SYSTEM
  
```

Listing B.7: Generated CSP_M model of Listing 6.7.

B.2.1.2 Bidirectional - Half-duplex

```

-----
-- Auto generated CSPM model by OpTrace
-- HalfDuplex
-- Input:
-- {P,Q}
-- {P,R}
-- {P,S}
-----

channel Ap,A,Bp,B,Cp,C,Aq,Br,Cs

aP = { | Ap,Bp,Cp | }
NodeP = ( Ap -> NodeP ) [ ] ( Bp -> NodeP ) [ ] ( Cp -> NodeP )

aQ = { | Aq | }
NodeQ = ( Aq -> NodeQ )

aR = { | Br | }
NodeR = ( Br -> NodeR )
  
```

```

aS = { | Cs | }
NodeS = ( Cs -> NodeS )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A)) [] (A -> (Ap -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (Bp -> (B -> ChanP_B [] ChanP_B)) [] (B -> (Bp -> ChanP_B [] ChanP_B))

aCPC = { | Cp, C | }
ChanP_C = (Cp -> (C -> ChanP_C [] ChanP_C)) [] (C -> (Cp -> ChanP_C [] ChanP_C))

aCQA = { | Aq, A | }
ChanQ_A = (Aq -> (A -> ChanQ_A [] ChanQ_A)) [] (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCRB = { | Br, B | }
ChanR_B = (Br -> (B -> ChanR_B [] ChanR_B)) [] (B -> (Br -> ChanR_B [] ChanR_B))

aCSC = { | Cs, C | }
ChanS_C = (Cs -> (C -> ChanS_C [] ChanS_C)) [] (C -> (Cs -> ChanS_C [] ChanS_C))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ) [] {Ap, Bp, Cp, Aq, Br, Cs} || ←
  (((((ChanP_A [ { |Ap,A| } || { |Bp,B| } ] ChanP_B) [ { |Ap,A,Bp,B| } || { |Cp,C| } ] ←
  ChanP_C) [ { |Ap,A,Bp,B,Cp,C| } || { |Aq,A| } ] ChanQ_A) [ { |Ap,A,Bp,B,Cp,C,Aq| } || ←
  { |Br,B| } ] ChanR_B) [ { |Ap,A,Bp,B,Cp,C,Aq,Br| } || { |Cs,C| } ] ChanS_C)
MAIN = SYSTEM
  
```

Listing B.8: Generated CSP_M model of Listing 6.8.

B.2.1.3 Bidirectional - Simplex

```

-----
-- Auto generated CSPM model by OpTrace
-- Simplex
-- Input:
-- (P,Q)
-- (P,R)
  
```

```

-- (P,S)
-- (Q,P)
-- (R,P)
-- (S,P)
-----

channel Ap, A, Bp, B, Cp, C, Dp, D, Ep, E, Fp, F, Aq, Dq, Br, Er, Cs, Fs

aP = { | Ap, Bp, Cp, Dp, Ep, Fp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP ) [] ( Cp -> NodeP ) [] ( Dp -> NodeP ) [] ( Ep ->
-> NodeP ) [] ( Fp -> NodeP )

aQ = { | Dq, Aq | }
NodeQ = ( Aq -> NodeQ ) [] ( Dq -> NodeQ )

aR = { | Er, Br | }
NodeR = ( Br -> NodeR ) [] ( Er -> NodeR )

aS = { | Fs, Cs | }
NodeS = ( Cs -> NodeS ) [] ( Fs -> NodeS )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (Bp -> (B -> ChanP_B [] ChanP_B))

aCPC = { | Cp, C | }
ChanP_C = (Cp -> (C -> ChanP_C [] ChanP_C))

aCPD = { | Dp, D | }
ChanP_D = (D -> (Dp -> ChanP_D [] ChanP_D))

aCPE = { | Ep, E | }
ChanP_E = (E -> (Ep -> ChanP_E [] ChanP_E))

aCPF = { | Fp, F | }

```

```

ChanP_F = (F -> (Fp -> ChanP_F [] ChanP_F))

aCQA = { | Aq, A | }
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQD = { | Dq, D | }
ChanQ_D = (Dq -> (D -> ChanQ_D [] ChanQ_D))

aCRB = { | Br, B | }
ChanR_B = (B -> (Br -> ChanR_B [] ChanR_B))

aCRE = { | Er, E | }
ChanR_E = (Er -> (E -> ChanR_E [] ChanR_E))

aCSC = { | Cs, C | }
ChanS_C = (C -> (Cs -> ChanS_C [] ChanS_C))

aCSF = { | Fs, F | }
ChanS_F = (Fs -> (F -> ChanS_F [] ChanS_F))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ) [ | {Ap, Bp, Cp, Dp, Ep, Fp, Aq, Dq, ←
  Br, Er, Cs, Fs} | | (((((((((((ChanP_A [ | {Ap,A} | | { |Bp,B} | ] ChanP_B) [ | {Ap, ←
  A,Bp,B} | | { |Cp,C} | ] ChanP_C) [ | {Ap,A,Bp,B,Cp,C} | | { |Dp,D} | ] ChanP_D) [ | { ←
  Ap,A,Bp,B,Cp,C,Dp,D} | | { |Ep,E} | ] ChanP_E) [ | {Ap,A,Bp,B,Cp,C,Dp,D,Ep,E} | | ←
  { |Fp,F} | ] ChanP_F) [ | {Ap,A,Bp,B,Cp,C,Dp,D,Ep,E,Fp,F} | | { |Aq,A} | ] ChanQ_A) [ ←
  { |Ap,A,Bp,B,Cp,C,Dp,D,Ep,E,Fp,F,Aq} | | { |Dq,D} | ] ChanQ_D) [ | {Ap,A,Bp,B,Cp,C, ←
  Dp,D,Ep,E,Fp,F,Aq,Dq} | | { |Br,B} | ] ChanR_B) [ | {Ap,A,Bp,B,Cp,C,Dp,D,Ep,E,Fp,F, ←
  Aq,Dq,Br} | | { |Er,E} | ] ChanR_E) [ | {Ap,A,Bp,B,Cp,C,Dp,D,Ep,E,Fp,F,Aq,Dq,Br,Er ←
  } | | { |Cs,C} | ] ChanS_C) [ | {Ap,A,Bp,B,Cp,C,Dp,D,Ep,E,Fp,F,Aq,Dq,Br,Er,Cs} | | ←
  { |Fs,F} | ] ChanS_F)
MAIN = SYSTEM

```

Listing B.9: Generated CSP_M model of Listing 6.9.

B.3 CHAIN TOPOLOGY

B.3.1 3-Node Ring

B.3.1.1 Broadcasting

```

-----
-- Auto generated CSPM model by OpTrace
-- Broadcasting
-- Input:
-- (P,Q,R)
-- (Q,P,R)
-- (R,P,Q)
-----

channel Ap, A, Bp, B, Cp, C, Aq, Bq, Cq, Ar, Br, Cr

aP = { | Ap, Bp, Cp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP ) [] ( Cp -> NodeP )

aQ = { | Bq, Aq, Cq | }
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ ) [] ( Cq -> NodeQ )

aR = { | Cr, Ar, Br | }
NodeR = ( Ar -> NodeR ) [] ( Br -> NodeR ) [] ( Cr -> NodeR )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (B -> (Bp -> ChanP_B [] ChanP_B))

aCPC = { | Cp, C | }
ChanP_C = (C -> (Cp -> ChanP_C [] ChanP_C))

aCQA = { | Aq, A | }
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

```

```

aCQB = { | Bq, B | }
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))

aCQC = { | Cq, C | }
ChanQ_C = (C -> (Cq -> ChanQ_C [] ChanQ_C))

aCRA = { | Ar, A | }
ChanR_A = (A -> (Ar -> ChanR_A [] ChanR_A))

aCRB = { | Br, B | }
ChanR_B = (B -> (Br -> ChanR_B [] ChanR_B))

aCRC = { | Cr, C | }
ChanR_C = (C -> (Cr -> ChanR_C [] ChanR_C))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ) [ | {Ap, Bp, Cp, Aq, Bq, Cq, Ar, Br, Cr} | ] ←
  (((((((ChanP_A [ | {Ap,A} | | {Bp,B} ] ChanP_B) [ | {Ap,A,Bp,B} | | {Cp,C} ] ←
  ChanP_C) [ | {Ap,A,Bp,B,Cp,C} | | {Aq,A} ] ChanQ_A) [ | {Ap,A,Bp,B,Cp,C,Aq} | | ←
  {Bq,B} ] ChanQ_B) [ | {Ap,A,Bp,B,Cp,C,Aq,Bq} | | {Cq,C} ] ChanQ_C) [ | {Ap,A, ←
  Bp,B,Cp,C,Aq,Bq,Cq} | | {Ar,A} ] ChanR_A) [ | {Ap,A,Bp,B,Cp,C,Aq,Bq,Cq,Ar} | | ←
  {Br,B} ] ChanR_B) [ | {Ap,A,Bp,B,Cp,C,Aq,Bq,Cq,Ar,Br} | | {Cr,C} ] ChanR_C)
MAIN = SYSTEM
  
```

Listing B.10: Generated CSP_M model of Listing 6.10.

B.3.1.2 Bidirectional - Half-duplex

```

-----
-- Auto generated CSPM model by OpTrace
-- HalfDuplex
-- Input:
-- {P,Q}
-- {P,R}
-- {Q,R}
-----

channel Ap, A, Bp, B, Aq, Cq, C, Br, Cr
  
```



```

aP = { | Ap,Bp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP )

aQ = { | Aq,Cq | }
NodeQ = ( Aq -> NodeQ ) [] ( Cq -> NodeQ )

aR = { | Br,Cr | }
NodeR = ( Br -> NodeR ) [] ( Cr -> NodeR )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A)) [] (A -> (Ap -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (Bp -> (B -> ChanP_B [] ChanP_B)) [] (B -> (Bp -> ChanP_B [] ChanP_B))

aCQA = { | Aq, A | }
ChanQ_A = (Aq -> (A -> ChanQ_A [] ChanQ_A)) [] (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQC = { | Cq, C | }
ChanQ_C = (Cq -> (C -> ChanQ_C [] ChanQ_C)) [] (C -> (Cq -> ChanQ_C [] ChanQ_C))

aCRB = { | Br, B | }
ChanR_B = (Br -> (B -> ChanR_B [] ChanR_B)) [] (B -> (Br -> ChanR_B [] ChanR_B))

aCRC = { | Cr, C | }
ChanR_C = (Cr -> (C -> ChanR_C [] ChanR_C)) [] (C -> (Cr -> ChanR_C [] ChanR_C))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ) [] { Ap, Bp, Aq, Cq, Br, Cr } [] (((((ChanP_A [←
    { |Ap,A| } || { |Bp,B| } ] ChanP_B) [ { |Ap,A,Bp,B| } || { |Aq,A| } ] ChanQ_A) [ { |Ap,A←
    ,Bp,B,Aq| } || { |Cq,C| } ] ChanQ_C) [ { |Ap,A,Bp,B,Aq,Cq,C| } || { |Br,B| } ] ChanR_B)←
    [ { |Ap,A,Bp,B,Aq,Cq,C,Br| } || { |Cr,C| } ] ChanR_C)
MAIN = SYSTEM
  
```

Listing B.11: Generated CSP_M model of Listing 6.11.

B.3.1.3 Unidirectional - Simplex

```

-----
-- Auto generated CSPM model by OpTrace
-- Simplex
-- Input:
-- (P,R)
-- (Q,P)
-- (R,Q)
-----

channel Ap,A,Bp,B,Ar,Cr,C,Bq,Cq

aP = { | Ap,Bp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP )

aR = { | Cr,Ar | }
NodeR = ( Ar -> NodeR ) [] ( Cr -> NodeR )

aQ = { | Bq,Cq | }
NodeQ = ( Bq -> NodeQ ) [] ( Cq -> NodeQ )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (B -> (Bp -> ChanP_B [] ChanP_B))

aCRA = { | Ar, A | }
ChanR_A = (A -> (Ar -> ChanR_A [] ChanR_A))

aCRC = { | Cr, C | }
ChanR_C = (Cr -> (C -> ChanR_C [] ChanR_C))

aQCB = { | Bq, B | }
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))

aCQC = { | Cq, C | }

```

```

ChanQ_C = ( C -> ( Cq -> ChanQ_C [] ChanQ_C ) )

SYSTEM = ( NodeP ||| NodeR ||| NodeQ ) [] { Ap, Bp, Ar, Cr, Bq, Cq } [] ( ((( ( ChanP_A [ <->
  { | Ap, A | } || { | Bp, B | } ] ChanP_B ) [ { | Ap, A, Bp, B | } || { | Ar, A | } ] ChanR_A ) [ { | Ap, A <->
  , Bp, B, Ar | } || { | Cr, C | } ] ChanR_C ) [ { | Ap, A, Bp, B, Ar, Cr, C | } || { | Bq, B | } ] ChanQ_B ) <->
  [ { | Ap, A, Bp, B, Ar, Cr, C, Bq | } || { | Cq, C | } ] ChanQ_C )
MAIN = SYSTEM
  
```

Listing B.12: Generated CSP_M model of Listing 6.12.

B.3.2 4-Node Ring

B.3.2.1 Broadcasting

```

-----
-- Auto generated CSPM model by OpTrace
-- Broadcasting
-- Input:
-- (P, Q, S)
-- (Q, P, R)
-- (R, Q, S)
-- (S, P, R)
-----

channel Ap, A, Bp, B, Dp, D, Aq, Bq, Cq, C, As, Cs, Ds, Br, Cr, Dr

aP = { | Ap, Bp, Dp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP ) [] ( Dp -> NodeP )

aQ = { | Bq, Aq, Cq | }
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ ) [] ( Cq -> NodeQ )

aS = { | Ds, As, Cs | }
NodeS = ( As -> NodeS ) [] ( Cs -> NodeS ) [] ( Ds -> NodeS )

aR = { | Cr, Br, Dr | }
NodeR = ( Br -> NodeR ) [] ( Cr -> NodeR ) [] ( Dr -> NodeR )
  
```

```
aCPA = { | Ap, A | }  
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))  
  
aCPB = { | Bp, B | }  
ChanP_B = (B -> (Bp -> ChanP_B [] ChanP_B))  
  
aCPD = { | Dp, D | }  
ChanP_D = (D -> (Dp -> ChanP_D [] ChanP_D))  
  
aCQA = { | Aq, A | }  
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))  
  
aCQB = { | Bq, B | }  
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))  
  
aCQC = { | Cq, C | }  
ChanQ_C = (C -> (Cq -> ChanQ_C [] ChanQ_C))  
  
aCSA = { | As, A | }  
ChanS_A = (A -> (As -> ChanS_A [] ChanS_A))  
  
aCSC = { | Cs, C | }  
ChanS_C = (C -> (Cs -> ChanS_C [] ChanS_C))  
  
aCSD = { | Ds, D | }  
ChanS_D = (Ds -> (D -> ChanS_D [] ChanS_D))  
  
aCRB = { | Br, B | }  
ChanR_B = (B -> (Br -> ChanR_B [] ChanR_B))  
  
aCRC = { | Cr, C | }  
ChanR_C = (C -> (Cr -> ChanR_C [] ChanR_C))  
  
aCRD = { | Dr, D | }  
ChanR_D = (D -> (Dr -> ChanR_D [] ChanR_D))
```

```

SYSTEM = ( NodeP ||| NodeQ ||| NodeS ||| NodeR ) [ | {Ap, Bp, Dp, Aq, Bq, Cq, As, Cs, ←
  Ds, Br, Cr, Dr} | | ((((((((((ChanP_A [ | {Ap,A} | | {Bp,B} ] ChanP_B) [ | {Ap, ←
  A,Bp,B} | | {Dp,D} ] ChanP_D) [ | {Ap,A,Bp,B,Dp,D} | | {Aq,A} ] ChanQ_A) [ | { ←
  Ap,A,Bp,B,Dp,D,Aq} | | {Bq,B} ] ChanQ_B) [ | {Ap,A,Bp,B,Dp,D,Aq,Bq} | | {Cq,C ←
  } ] ChanQ_C) [ | {Ap,A,Bp,B,Dp,D,Aq,Bq,Cq,C} | | {As,A} ] ChanS_A) [ | {Ap,A,Bp ←
  ,B,Dp,D,Aq,Bq,Cq,C,As} | | {Cs,C} ] ChanS_C) [ | {Ap,A,Bp,B,Dp,D,Aq,Bq,Cq,C,As, ←
  Cs} | | {Ds,D} ] ChanS_D) [ | {Ap,A,Bp,B,Dp,D,Aq,Bq,Cq,C,As,Cs,Ds} | | {Br,B} ←
  ] ChanR_B) [ | {Ap,A,Bp,B,Dp,D,Aq,Bq,Cq,C,As,Cs,Ds,Br} | | {Cr,C} ] ChanR_C) [ ←
  {Ap,A,Bp,B,Dp,D,Aq,Bq,Cq,C,As,Cs,Ds,Br,Cr} | | {Dr,D} ] ChanR_D)
MAIN = SYSTEM
  
```

Listing B.13: Generated CSP_M model of Listing 6.13.

B.3.2.2 Bidirectional - Half-duplex

```

-----
-- Auto generated CSPM model by OpTrace
-- HalfDuplex
-- Input:
-- {P,Q}
-- {P,S}
-- {Q,R}
-- {R,S}
-----

channel Ap,A,Bp,B,Aq,Cq,C,Bs,Ds,D,Cr,Dr

aP = { | Ap,Bp | }
NodeP = ( Ap -> NodeP ) [ | ( Bp -> NodeP )

aQ = { | Aq,Cq | }
NodeQ = ( Aq -> NodeQ ) [ | ( Cq -> NodeQ )

aS = { | Bs,Ds | }
NodeS = ( Bs -> NodeS ) [ | ( Ds -> NodeS )

aR = { | Cr,Dr | }
NodeR = ( Cr -> NodeR ) [ | ( Dr -> NodeR )
  
```

```

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A)) [] (A -> (Ap -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (Bp -> (B -> ChanP_B [] ChanP_B)) [] (B -> (Bp -> ChanP_B [] ChanP_B))

aCQA = { | Aq, A | }
ChanQ_A = (Aq -> (A -> ChanQ_A [] ChanQ_A)) [] (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQC = { | Cq, C | }
ChanQ_C = (Cq -> (C -> ChanQ_C [] ChanQ_C)) [] (C -> (Cq -> ChanQ_C [] ChanQ_C))

aCSB = { | Bs, B | }
ChanS_B = (Bs -> (B -> ChanS_B [] ChanS_B)) [] (B -> (Bs -> ChanS_B [] ChanS_B))

aCSD = { | Ds, D | }
ChanS_D = (Ds -> (D -> ChanS_D [] ChanS_D)) [] (D -> (Ds -> ChanS_D [] ChanS_D))

aCRC = { | Cr, C | }
ChanR_C = (Cr -> (C -> ChanR_C [] ChanR_C)) [] (C -> (Cr -> ChanR_C [] ChanR_C))

aCRD = { | Dr, D | }
ChanR_D = (Dr -> (D -> ChanR_D [] ChanR_D)) [] (D -> (Dr -> ChanR_D [] ChanR_D))

SYSTEM = ( NodeP ||| NodeQ ||| NodeS ||| NodeR ) [ | {Ap, Bp, Aq, Cq, Bs, Ds, Cr, Dr} ←
  | | ((((((ChanP_A [ | {Ap,A} | | { |Bp,B} | ] ChanP_B) [ | {Ap,A,Bp,B} | | { |Aq,A} ←
  ] ChanQ_A) [ | {Ap,A,Bp,B,Aq} | | { |Cq,C} | ] ChanQ_C) [ | {Ap,A,Bp,B,Aq,Cq,C} | | ←
  { |Bs,B} | ] ChanS_B) [ | {Ap,A,Bp,B,Aq,Cq,C,Bs} | | { |Ds,D} | ] ChanS_D) [ | {Ap,A, ←
  Bp,B,Aq,Cq,C,Bs,Ds,D} | | { |Cr,C} | ] ChanR_C) [ | {Ap,A,Bp,B,Aq,Cq,C,Bs,Ds,D,Cr} ←
  | | { |Dr,D} | ] ChanR_D)
MAIN = SYSTEM

```

Listing B.14: Generated CSP_M model of Listing 6.14.

B.3.2.3 Unidirectional - Simplex

```

-----
-- Auto generated CSPM model by OpTrace
-- Simplex
-- Input:
-- (P,S)
-- (Q,R)
-- (R,Q)
-- (S,R)
-----

channel Ap,A,As,Ds,D,Bq,B,Cq,C,Br,Cr,Dr

aP = { | Ap | }
NodeP = ( Ap -> NodeP )

aS = { | Ds,As | }
NodeS = ( As -> NodeS ) [] ( Ds -> NodeS )

aQ = { | Bq,Cq | }
NodeQ = ( Bq -> NodeQ ) [] ( Cq -> NodeQ )

aR = { | Cr,Br,Dr | }
NodeR = ( Br -> NodeR ) [] ( Cr -> NodeR ) [] ( Dr -> NodeR )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCSA = { | As, A | }
ChanS_A = (A -> (As -> ChanS_A [] ChanS_A))

aCSD = { | Ds, D | }
ChanS_D = (Ds -> (D -> ChanS_D [] ChanS_D))

aCQB = { | Bq, B | }
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))
  
```

```

aCQC = { | Cq, C | }
ChanQ_C = ( C -> ( Cq -> ChanQ_C [] ChanQ_C ) )

aCRB = { | Br, B | }
ChanR_B = ( B -> ( Br -> ChanR_B [] ChanR_B ) )

aCRC = { | Cr, C | }
ChanR_C = ( Cr -> ( C -> ChanR_C [] ChanR_C ) )

aCRD = { | Dr, D | }
ChanR_D = ( D -> ( Dr -> ChanR_D [] ChanR_D ) )

SYSTEM = ( NodeP ||| NodeS ||| NodeQ ||| NodeR ) [ | { Ap, As, Ds, Bq, Cq, Br, Cr, Dr } <->
  | ] ( ( ( ( ( ( ChanP_A [ | { Ap, A } | | { As, A } ] ChanS_A [ | { Ap, A, As } | | { | Ds, D | } ] <->
  ChanS_D [ | { Ap, A, As, Ds, D } | | { | Bq, B | } ] ChanQ_B [ | { Ap, A, As, Ds, D, Bq, B } | | <->
  { | Cq, C | } ] ChanQ_C [ | { Ap, A, As, Ds, D, Bq, B, Cq, C } | | { | Br, B | } ] ChanR_B [ | { Ap, A <->
  , As, Ds, D, Bq, B, Cq, C, Br } | | { | Cr, C | } ] ChanR_C [ | { Ap, A, As, Ds, D, Bq, B, Cq, C, Br, Cr <->
  | } | | { | Dr, D | } ] ChanR_D )
MAIN = SYSTEM
  
```

Listing B.15: Generated CSP_M model of Listing 6.15.

B.3.3 4-Node Linear

B.3.3.1 Broadcasting

```

-----
-- Auto generated CSPM model by OpTrace
-- Broadcasting
-- Input:
-- (P,Q)
-- (Q,P,R)
-- (R,Q,S)
-- (S,R)
-----

channel Ap,A,Bp,B,Aq,Bq,Cq,C,Br,Cr,Dr,D,Cs,Ds
  
```



```

aP = { | Ap,Bp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP )

aQ = { | Bq,Aq,Cq | }
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ ) [] ( Cq -> NodeQ )

aR = { | Cr,Br,Dr | }
NodeR = ( Br -> NodeR ) [] ( Cr -> NodeR ) [] ( Dr -> NodeR )

aS = { | Ds,Cs | }
NodeS = ( Cs -> NodeS ) [] ( Ds -> NodeS )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (B -> (Bp -> ChanP_B [] ChanP_B))

aCQA = { | Aq, A | }
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQB = { | Bq, B | }
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))

aCQC = { | Cq, C | }
ChanQ_C = (C -> (Cq -> ChanQ_C [] ChanQ_C))

aCRB = { | Br, B | }
ChanR_B = (B -> (Br -> ChanR_B [] ChanR_B))

aCRC = { | Cr, C | }
ChanR_C = (Cr -> (C -> ChanR_C [] ChanR_C))

aCRD = { | Dr, D | }
ChanR_D = (D -> (Dr -> ChanR_D [] ChanR_D))

aCSC = { | Cs, C | }

```

```

ChanS_C = ( C -> (Cs -> ChanS_C [] ChanS_C)

aCSD = { | Ds, D | }

ChanS_D = (Ds -> (D -> ChanS_D [] ChanS_D))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ) [ | {Ap, Bp, Aq, Bq, Cq, Br, Cr, Dr, ←
  Cs, Ds} | ] ( ( ( ( ( ( ( ( ChanP_A [ | {Ap, A} | | | {Bp, B} | ] ChanP_B) [ | {Ap, A, Bp, B} | ←
  | | | {Aq, A} | ] ChanQ_A) [ | {Ap, A, Bp, B, Aq} | | | {Bq, B} | ] ChanQ_B) [ | {Ap, A, Bp, B, ←
  Aq, Bq} | | | {Cq, C} | ] ChanQ_C) [ | {Ap, A, Bp, B, Aq, Bq, Cq, C} | | | {Br, B} | ] ChanR_B) ←
  [ | {Ap, A, Bp, B, Aq, Bq, Cq, C, Br} | | | {Cr, C} | ] ChanR_C) [ | {Ap, A, Bp, B, Aq, Bq, Cq, C, ←
  Br, Cr} | | | {Dr, D} | ] ChanR_D) [ | {Ap, A, Bp, B, Aq, Bq, Cq, C, Br, Cr, Dr, D} | | | {Cs, C} | ←
  ] ChanS_C) [ | {Ap, A, Bp, B, Aq, Bq, Cq, C, Br, Cr, Dr, D, Cs} | | | {Ds, D} | ] ChanS_D)

MAIN = SYSTEM
  
```

Listing B.16: Generated CSP_M model of Listing 6.16.

B.3.3.2 Bidirectional - Half-duplex

```

-----
-- Auto generated CSPM model by OpTrace
-- HalfDuplex
-- Input:
-- {P,Q}
-- {Q,R}
-- {R,S}
-----

channel Ap,A,Aq,Bq,B,Br,Cr,C,Cs

aP = { | Ap | }
NodeP = ( Ap -> NodeP )

aQ = { | Aq,Bq | }
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ )

aR = { | Br,Cr | }
NodeR = ( Br -> NodeR ) [] ( Cr -> NodeR )
  
```

```

aS = { | Cs | }
NodeS = ( Cs -> NodeS )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A)) [] (A -> (Ap -> ChanP_A [] ChanP_A))

aCQA = { | Aq, A | }
ChanQ_A = (Aq -> (A -> ChanQ_A [] ChanQ_A)) [] (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQB = { | Bq, B | }
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B)) [] (B -> (Bq -> ChanQ_B [] ChanQ_B))

aCRB = { | Br, B | }
ChanR_B = (Br -> (B -> ChanR_B [] ChanR_B)) [] (B -> (Br -> ChanR_B [] ChanR_B))

aCRC = { | Cr, C | }
ChanR_C = (Cr -> (C -> ChanR_C [] ChanR_C)) [] (C -> (Cr -> ChanR_C [] ChanR_C))

aCSC = { | Cs, C | }
ChanS_C = (Cs -> (C -> ChanS_C [] ChanS_C)) [] (C -> (Cs -> ChanS_C [] ChanS_C))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ) [] {Ap, Aq, Bq, Br, Cr, Cs} || ←
  (((((ChanP_A [ { |Ap,A| } || { |Aq,A| } ] ChanQ_A) [ { |Ap,A,Aq| } || { |Bq,B| } ] ←
  ChanQ_B) [ { |Ap,A,Aq,Bq,B| } || { |Br,B| } ] ChanR_B) [ { |Ap,A,Aq,Bq,B,Br| } || { |Cr←
  ,C| } ] ChanR_C) [ { |Ap,A,Aq,Bq,B,Br,Cr,C| } || { |Cs,C| } ] ChanS_C)
MAIN = SYSTEM
  
```

Listing B.17: Generated CSP_M model of Listing 6.17.

B.3.3.3 Unidirectional - Simplex

```

-----
-- Auto generated CSPM model by OpTrace
-- Simplex
-- Input:
-- (P,Q)
-- (Q,R)
-- (R,S)
  
```

```
channel Ap, A, Aq, Bq, B, Br, Cr, C, Cs
```

```
aP = { | Ap | }
```

```
NodeP = ( Ap -> NodeP )
```

```
aQ = { | Bq, Aq | }
```

```
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ )
```

```
aR = { | Cr, Br | }
```

```
NodeR = ( Br -> NodeR ) [] ( Cr -> NodeR )
```

```
aS = { | Cs | }
```

```
NodeS = ( Cs -> NodeS )
```

```
aCPA = { | Ap, A | }
```

```
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))
```

```
aCQA = { | Aq, A | }
```

```
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))
```

```
aCQB = { | Bq, B | }
```

```
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))
```

```
aCRB = { | Br, B | }
```

```
ChanR_B = (B -> (Br -> ChanR_B [] ChanR_B))
```

```
aCRC = { | Cr, C | }
```

```
ChanR_C = (Cr -> (C -> ChanR_C [] ChanR_C))
```

```
aCSC = { | Cs, C | }
```

```
ChanS_C = (C -> (Cs -> ChanS_C [] ChanS_C))
```

```
SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ) [] {Ap, Aq, Bq, Br, Cr, Cs} || ←  
(((ChanP_A [ { |Ap, A| } || { |Aq, A| } ] ChanQ_A) [ { |Ap, A, Aq| } || { |Bq, B| } ] ←  
ChanQ_B) [ { |Ap, A, Aq, Bq, B| } || { |Br, B| } ] ChanR_B) [ { |Ap, A, Aq, Bq, B, Br| } || { |Cr, C| } ]
```

```

    ,C|} ] ChanR_C) [ { |Ap,A,Aq,Bq,B,Br,Cr,C|} || { |Cs,C|} ] ChanS_C)
MAIN = SYSTEM
  
```

Listing B.18: Generated CSP_M model of Listing 6.18.

B.4 TREE TOPOLOGY

B.4.1 7-Node Tree

B.4.1.1 Broadcasting

```

-----
-- Auto generated CSPM model by OpTrace
-- Broadcasting
-- Input:
-- (P,Q,R)
-- (Q,S,T)
-- (R,U,V)
-----

channel Ap,A,Aq,Bq,B,Ar,Cr,C,Bs,Bt,Cu,Cv

aP = { | Ap | }
NodeP = ( Ap -> NodeP )

aQ = { | Bq,Aq | }
NodeQ = ( Aq -> NodeQ ) [] ( Bq -> NodeQ )

aR = { | Cr,Ar | }
NodeR = ( Ar -> NodeR ) [] ( Cr -> NodeR )

aS = { | Bs | }
NodeS = ( Bs -> NodeS )

aT = { | Bt | }
NodeT = ( Bt -> NodeT )
  
```

```

aU = { | Cu | }
NodeU = ( Cu -> NodeU )

aV = { | Cv | }
NodeV = ( Cv -> NodeV )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A))

aCQA = { | Aq, A | }
ChanQ_A = (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQB = { | Bq, B | }
ChanQ_B = (Bq -> (B -> ChanQ_B [] ChanQ_B))

aCRA = { | Ar, A | }
ChanR_A = (A -> (Ar -> ChanR_A [] ChanR_A))

aCRC = { | Cr, C | }
ChanR_C = (Cr -> (C -> ChanR_C [] ChanR_C))

aCSB = { | Bs, B | }
ChanS_B = (B -> (Bs -> ChanS_B [] ChanS_B))

aCTB = { | Bt, B | }
ChanT_B = (B -> (Bt -> ChanT_B [] ChanT_B))

aCUC = { | Cu, C | }
ChanU_C = (C -> (Cu -> ChanU_C [] ChanU_C))

aCVC = { | Cv, C | }
ChanV_C = (C -> (Cv -> ChanV_C [] ChanV_C))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ||| NodeT ||| NodeU ||| NodeV ) [ | {←
    Ap, Aq, Bq, Ar, Cr, Bs, Bt, Cu, Cv} | ] ( ( ( ( ( ( ( ( ChanP_A [ { |Ap,A| } || { |Aq,A| } ] ←
    ChanQ_A [ { |Ap,A,Aq| } || { |Bq,B| } ] ChanQ_B [ { |Ap,A,Aq,Bq,B| } || { |Ar,A| } ] ←
    ChanR_A [ { |Ap,A,Aq,Bq,B,Ar| } || { |Cr,C| } ] ChanR_C [ { |Ap,A,Aq,Bq,B,Ar,Cr,C| } ←

```

```

    || { |Bs,B| } ] ChanS_B) [ { |Ap,A,Aq,Bq,B,Ar,Cr,C,Bs| } || { |Bt,B| } ] ChanT_B) [ ↔
    { |Ap,A,Aq,Bq,B,Ar,Cr,C,Bs,Bt| } || { |Cu,C| } ] ChanU_C) [ { |Ap,A,Aq,Bq,B,Ar,Cr,C,↔
    Bs,Bt,Cu| } || { |Cv,C| } ] ChanV_C)
  MAIN = SYSTEM

```

Listing B.19: Generated CSP_M model of Listing 6.19.

B.4.1.2 Bidirectional - Half-duplex

```

-----
-- Auto generated CSPM model by OpTrace
-- HalfDuplex
-- Input:
-- {P,Q}
-- {P,R}
-- {Q,S}
-- {Q,T}
-- {R,U}
-- {R,V}
-----

channel Ap,A,Bp,B,Aq,Cq,C,Dq,D,Br,Er,E,Fr,F,Cs,Dt,Eu,Fv

aP = { | Ap,Bp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP )

aQ = { | Aq,Cq,Dq | }
NodeQ = ( Aq -> NodeQ ) [] ( Cq -> NodeQ ) [] ( Dq -> NodeQ )

aR = { | Br,Er,Fr | }
NodeR = ( Br -> NodeR ) [] ( Er -> NodeR ) [] ( Fr -> NodeR )

aS = { | Cs | }
NodeS = ( Cs -> NodeS )

aT = { | Dt | }
NodeT = ( Dt -> NodeT )

```

```

aU = { | Eu | }
NodeU = ( Eu -> NodeU )

aV = { | Fv | }
NodeV = ( Fv -> NodeV )

aCPA = { | Ap, A | }
ChanP_A = (Ap -> (A -> ChanP_A [] ChanP_A)) [] (A -> (Ap -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (Bp -> (B -> ChanP_B [] ChanP_B)) [] (B -> (Bp -> ChanP_B [] ChanP_B))

aCQA = { | Aq, A | }
ChanQ_A = (Aq -> (A -> ChanQ_A [] ChanQ_A)) [] (A -> (Aq -> ChanQ_A [] ChanQ_A))

aCQC = { | Cq, C | }
ChanQ_C = (Cq -> (C -> ChanQ_C [] ChanQ_C)) [] (C -> (Cq -> ChanQ_C [] ChanQ_C))

aCQD = { | Dq, D | }
ChanQ_D = (Dq -> (D -> ChanQ_D [] ChanQ_D)) [] (D -> (Dq -> ChanQ_D [] ChanQ_D))

aCRB = { | Br, B | }
ChanR_B = (Br -> (B -> ChanR_B [] ChanR_B)) [] (B -> (Br -> ChanR_B [] ChanR_B))

aCRE = { | Er, E | }
ChanR_E = (Er -> (E -> ChanR_E [] ChanR_E)) [] (E -> (Er -> ChanR_E [] ChanR_E))

aCRF = { | Fr, F | }
ChanR_F = (Fr -> (F -> ChanR_F [] ChanR_F)) [] (F -> (Fr -> ChanR_F [] ChanR_F))

aCSC = { | Cs, C | }
ChanS_C = (Cs -> (C -> ChanS_C [] ChanS_C)) [] (C -> (Cs -> ChanS_C [] ChanS_C))

aCTD = { | Dt, D | }
ChanT_D = (Dt -> (D -> ChanT_D [] ChanT_D)) [] (D -> (Dt -> ChanT_D [] ChanT_D))

aCUE = { | Eu, E | }

```



```

ChanU_E = (Eu -> (E -> ChanU_E [] ChanU_E)) [] (E -> (Eu -> ChanU_E [] ChanU_E))

aCVF = { | Fv, F | }

ChanV_F = (Fv -> (F -> ChanV_F [] ChanV_F)) [] (F -> (Fv -> ChanV_F [] ChanV_F))

SYSTEM = ( NodeP ||| NodeQ ||| NodeR ||| NodeS ||| NodeT ||| NodeU ||| NodeV ) [ | { <-
  Ap, Bp, Aq, Cq, Dq, Br, Er, Fr, Cs, Dt, Eu, Fv } | ] ( ( ( ( ( ( ( ( ( ( ( ChanP_A [ { | Ap, A | } <-
  | | { | Bp, B | } ] ChanP_B) [ { | Ap, A, Bp, B | } | | { | Aq, A | } ] ChanQ_A) [ { | Ap, A, Bp, B, Aq <-
  | } | | { | Cq, C | } ] ChanQ_C) [ { | Ap, A, Bp, B, Aq, Cq, C | } | | { | Dq, D | } ] ChanQ_D) [ { | Ap, <-
  A, Bp, B, Aq, Cq, C, Dq, D | } | | { | Br, B | } ] ChanR_B) [ { | Ap, A, Bp, B, Aq, Cq, C, Dq, D, Br | } | | <-
  { | Er, E | } ] ChanR_E) [ { | Ap, A, Bp, B, Aq, Cq, C, Dq, D, Br, Er, E | } | | { | Fr, F | } ] ChanR_F) <-
  [ { | Ap, A, Bp, B, Aq, Cq, C, Dq, D, Br, Er, E, Fr, F | } | | { | Cs, C | } ] ChanS_C) [ { | Ap, A, Bp, B, <-
  Aq, Cq, C, Dq, D, Br, Er, E, Fr, F, Cs | } | | { | Dt, D | } ] ChanT_D) [ { | Ap, A, Bp, B, Aq, Cq, C, Dq, D <-
  , Br, Er, E, Fr, F, Cs, Dt | } | | { | Eu, E | } ] ChanU_E) [ { | Ap, A, Bp, B, Aq, Cq, C, Dq, D, Br, Er, E, <-
  Fr, F, Cs, Dt, Eu | } | | { | Fv, F | } ] ChanV_F)

MAIN = SYSTEM

```

Listing B.20: Generated CSP_M model of Listing 6.20.

B.4.1.3 Unidirectional - Simplex

```

-----
-- Auto generated CSPM model by OpTrace
-- Simplex
-- Input:
-- (Q,P)
-- (R,P)
-- (S,Q)
-- (T,Q)
-- (U,R)
-- (V,R)
-----

channel Aq, A, Cq, C, Dq, D, Ap, Bp, B, Br, Er, E, Fr, F, Cs, Dt, Eu, Fv

aQ = { | Aq, Cq, Dq | }

NodeQ = ( Aq -> NodeQ ) [ ] ( Cq -> NodeQ ) [ ] ( Dq -> NodeQ )

```

```

aP = { | Ap,Bp | }
NodeP = ( Ap -> NodeP ) [] ( Bp -> NodeP )

aR = { | Br,Er,Fr | }
NodeR = ( Br -> NodeR ) [] ( Er -> NodeR ) [] ( Fr -> NodeR )

aS = { | Cs | }
NodeS = ( Cs -> NodeS )

aT = { | Dt | }
NodeT = ( Dt -> NodeT )

aU = { | Eu | }
NodeU = ( Eu -> NodeU )

aV = { | Fv | }
NodeV = ( Fv -> NodeV )

aCQA = { | Aq, A | }
ChanQ_A = (Aq -> (A -> ChanQ_A [] ChanQ_A))

aCQC = { | Cq, C | }
ChanQ_C = (C -> (Cq -> ChanQ_C [] ChanQ_C))

aCQD = { | Dq, D | }
ChanQ_D = (D -> (Dq -> ChanQ_D [] ChanQ_D))

aCPA = { | Ap, A | }
ChanP_A = (A -> (Ap -> ChanP_A [] ChanP_A))

aCPB = { | Bp, B | }
ChanP_B = (B -> (Bp -> ChanP_B [] ChanP_B))

aCRB = { | Br, B | }
ChanR_B = (Br -> (B -> ChanR_B [] ChanR_B))

aCRE = { | Er, E | }

```

```

ChanR_E = (E -> (Er -> ChanR_E [] ChanR_E))

aCRF = { | Fr, F | }
ChanR_F = (F -> (Fr -> ChanR_F [] ChanR_F))

aCSC = { | Cs, C | }
ChanS_C = (Cs -> (C -> ChanS_C [] ChanS_C))

aCTD = { | Dt, D | }
ChanT_D = (Dt -> (D -> ChanT_D [] ChanT_D))

aCUE = { | Eu, E | }
ChanU_E = (Eu -> (E -> ChanU_E [] ChanU_E))

aCVF = { | Fv, F | }
ChanV_F = (Fv -> (F -> ChanV_F [] ChanV_F))

SYSTEM = ( NodeQ ||| NodeP ||| NodeR ||| NodeS ||| NodeT ||| NodeU ||| NodeV ) [ | {←
  Aq, Cq, Dq, Ap, Bp, Br, Er, Fr, Cs, Dt, Eu, Fv} | | (((((((((((ChanQ_A [ { |Aq,A|}←
  | | { |Cq,C| } ] ChanQ_C) [ { |Aq,A,Cq,C| } | | { |Dq,D| } ] ChanQ_D) [ { |Aq,A,Cq,C,Dq,←
  D| } | | { |Ap,A| } ] ChanP_A) [ { |Aq,A,Cq,C,Dq,D,Ap| } | | { |Bp,B| } ] ChanP_B) [ { |Aq←
  ,A,Cq,C,Dq,D,Ap,Bp,B| } | | { |Br,B| } ] ChanR_B) [ { |Aq,A,Cq,C,Dq,D,Ap,Bp,B,Br| } | |←
  { |Er,E| } ] ChanR_E) [ { |Aq,A,Cq,C,Dq,D,Ap,Bp,B,Br,Er,E| } | | { |Fr,F| } ] ChanR_F)←
  [ { |Aq,A,Cq,C,Dq,D,Ap,Bp,B,Br,Er,E,Fr,F| } | | { |Cs,C| } ] ChanS_C) [ { |Aq,A,Cq,C,←
  Dq,D,Ap,Bp,B,Br,Er,E,Fr,F,Cs| } | | { |Dt,D| } ] ChanT_D) [ { |Aq,A,Cq,C,Dq,D,Ap,Bp,B←
  ,Br,Er,E,Fr,F,Cs,Dt| } | | { |Eu,E| } ] ChanU_E) [ { |Aq,A,Cq,C,Dq,D,Ap,Bp,B,Br,Er,E,←
  Fr,F,Cs,Dt,Eu| } | | { |Fv,F| } ] ChanV_F)

MAIN = SYSTEM

```

Listing B.21: Generated CSP_M model of Listing 6.21.