

COMPARATIVE STUDIES, FORMAL SEMANTICS AND PVS ENCODING OF CSP#

SHI LING

NATIONAL UNIVERSITY OF SINGAPORE

2014

**COMPARATIVE STUDIES, FORMAL SEMANTICS AND
PVS ENCODING OF CSP#**

SHI LING

(B.Eng., East China Normal University (China), 2009)

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

A handwritten signature in black ink, reading "Shi Ling". The signature is written in a cursive, flowing style. The first name "Shi" is written with a capital 'S' and a lowercase 'h' and 'i'. The second name "Ling" is written with a capital 'L' and lowercase 'i', 'n', and 'g'. The signature is positioned above a horizontal line.

Shi Ling
12 August 2014

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to those who assisted me, in one way or another, with my Ph.D study in the past five years.

First and foremost, I am deeply indebted to my supervisor Dr. Dong Jin Song for his guidance, encouragement and insight throughout the course of my doctoral program. His careful reading and constructive criticism of early drafts and many other works made this thesis possible.

I am grateful to my mentors, Dr. Sun Jun and Dr. Liu Yang, for their valuable suggestions and comments on my research works. I am also thankful to Dr. Qin Shengchao and Dr. Zhao Yongxin for their research collaborations.

I would like to thank my thesis committee, Dr. Chin Wei Ngan and Dr. Hugh Anderson, who have provided constructive feedback through GRP to this final thesis. My thanks also goes to anonymous referees who have reviewed and provided valuable comments to previous publications that are parts of this thesis.

To my labmates, thank you for your support and friendship, for the stimulating discussions, and for all the fun we have had along the way.

I sincerely thank my parents Shi Jianhong and Gu Lanfeng, and my aunt Shi Juying for their love, encouragement and support in my years of study.

Finally and most importantly, I would like to express my appreciation to my beloved husband Chen Chunqing for his ceaseless love, encouragement, and support.

Contents

List of Tables	i
1 Introduction and Overview	1
1.1 Motivation and Goals	1
1.2 Thesis Outline	5
1.3 Acknowledgement of Published Work	6
2 Background	9
2.1 The CSP# Language	9
2.1.1 Syntax	10
2.1.2 Concurrency	12
2.1.3 A CSP# Example - the Peg Solitaire Game	12
2.2 UTP Theory	15
2.3 Prototype Verification System	17
3 Comparison of CSP Extensions and Tools	21
3.1 CSP _M vs. CSP#: Syntax	22
3.1.1 Data Perspective	23
3.1.2 Process Perspective	25
3.2 CSP _M vs. CSP#: Operational Semantics	27
3.2.1 SKIP	28

3.2.2	<i>CHAOS</i>	28
3.2.3	Channel Communication	29
3.2.4	Shared variables	32
3.2.5	Parallel composition	34
3.2.6	Interleaving	37
3.2.7	General choice	38
3.2.8	Conditional choice	38
3.2.9	Renaming	39
3.2.10	Untimed timeout	40
3.2.11	Discussion	40
3.3	Verification Tool Support	41
3.3.1	Verification	41
3.3.2	Experiment	43
3.3.3	Discussion	47
3.4	Summary	47
4	A UTP Semantics for CSP#	49
4.1	Denotational Semantics of CSP#	51
4.1.1	Semantic Model	51
4.1.2	Semantics of Expressions and Programs	55
4.1.3	Semantics of Processes	56
4.2	Algebraic Laws	68
4.2.1	State Guard	68
4.2.2	Sequential Composition	69
4.2.3	Parallel Composition	70
4.3	The Closed Semantics	79
4.4	Summary	81

5	Encoding CSP# Denotational Semantics into PVS	85
5.1	The Theory of Semantic Model	87
5.1.1	The Theory of Observational Variables	87
5.1.2	The Theory of Healthiness Conditions	89
5.2	The Theories of Expressions and Programs	91
5.3	The Theory of Processes	95
5.3.1	Primitives	95
5.3.2	Sequential Composition	96
5.3.3	Event prefixing	97
5.3.4	Synchronous Channel Output/Input	97
5.3.5	Data Operation Prefixing	99
5.3.6	Choice	99
5.3.7	State Guard	100
5.3.8	Parallel Composition	101
5.3.9	Interleave	107
5.3.10	Hiding	107
5.3.11	Refinement	109
5.3.12	Recursion	109
5.4	Mechanical Proof of Laws	110
5.5	Summary	112
6	Conclusion	115
6.1	Contributions	115
6.2	Future Work	117
	Bibliography	119

Appendix A	CSP# Models for Benchmark Systems	127
A.1	Readers-writers Problem	127
A.2	Dinning Philosopher	128
A.3	Milner’s Cyclic Scheduler	129
A.4	The Peg Solitaire Game	129
A.5	Knight’s Tour	131
A.6	The Tower of Hanoi Puzzle	132
A.7	Concurrent Stack	133
A.8	Peterson’s Algorithm	135
Appendix B	Monotonicity of CSP# Process Combinators	137

Summary

Concurrency becomes an important and necessary property of large and complex systems. Many concurrent systems feature various interactions between execution processes, which are often communications via synchronous/asynchronous message passing or through shared resources. The intricate execution nature and common mission-critical feature of concurrent systems demand rigorous modelling and analysis methods at the early system design stage.

Communicating Sequential Processes (CSP) is a well-known formal specification language to model and analyse concurrent systems. Considerable efforts have been made to extend CSP to support emerging system features like data aspects by integrating declarative specification languages like Z, although the resulting CSP extensions lack automated analysis support.

Recently, Communicating Sequential Programs (CSP#) has been proposed to integrate high-level CSP-like process operators with low-level program constructs on the shared variables. Although these CSP-like extensions support similar types of concurrent systems, there are subtle and substantial differences between them, not only modelling features, but also tool support and verifiability. Our first work is to conduct comprehensive comparisons between CSP# and CSP_M (a noticeable CSP extension) from the perspectives of operational semantics and verification capabilities together with eight benchmark systems. These comparisons provide insights for users to select suitable languages/tools for various concurrent systems.

CSP# operational semantics has been defined and used in its PAT model checker. However, it is not compositional, and lacks the support of compositional verification. Our second work is to propose a compositional denotational semantics of CSP# using the Unifying Theories of Programming (UTP). Our denotational semantics blends communication events with state transitions containing shared variables, and captures all possible concurrency behaviours. It also considers the interference of the environment to process behaviours. We further define a set of algebraic laws capturing the distinct features of CSP#.

Proving our defined algebraic laws is important as such proofs can validate the correctness of the CSP# denotational semantics, although manual proving is tedious and subtle mistakes can easily occur. Moreover, a high grade of automated verification can save much human effort. Therefore, our third work is to encode CSP# denotational semantics into the Prototype Verification System (PVS), an integrated framework for formal specification and verification. Our encoding not only checks the semantics consistency, but also builds up a theoretic foundation for mechanical verification of CSP# models.

Key words: Concurrency, Shared Variables, Denotational Semantics, UTP, Encoding, PVS

List of Tables

3.1	Similar syntax among CSP, CSP_M and $\text{CSP}_\#$	23
3.2	Experiment results on refinement checking	44
3.3	Experiment results on solving puzzles	45
3.4	Experiment results on shared variables	46
3.5	Experiment results on LTL checking	46
5.1	Predicate formalisation in PVS	89
5.2	$\text{CSP}_\#$ process syntax	96

Chapter 1

Introduction and Overview

1.1 Motivation and Goals

Concurrency becomes an important and necessary property of widespread distributed systems such as networks of bank ATMs and Hadoop Mapreduce framework. Many concurrent systems feature various interactions between execution processes, which are often communications via synchronous/asynchronous message passing or through shared resources. The intricate execution nature and commonly mission-critical feature of such concurrent systems demand rigorous modelling and analysis methods at the early system design stage. One approach is applying formal methods that are mathematically based languages, techniques, and tools for the specification, development and verification of these concurrent systems [17]. This approach has been increasingly adopted by industry [96]. For example, Microsoft Hyper-V Hypervisor for virtualization (a technique allowing multiple guest operating systems to run concurrently on a single hardware platform) has been formally verified to prevent the risk of malicious attack [18].

Formal specification languages, as the basis for applying formal methods, are used to model

high-level system behaviour and properties. There are mainly two groups of formal specification languages: one has focused on the data aspects of the system, for example, Z [81], VDM (Vienna Development Method) [36] and B [2], while the other has emphasized on the behaviour of the system, such as CSP (Communicating Sequential Processes) [29] and CCS (Calculus of Communicating Systems) [49]. A language from one group can express only a particular characteristic of a system. For example, Z notation, based on the set theory and predicate logic, is used to specify the data states of the system; however, the associated data operations are modelled in a declarative style, and cannot be executed. On the other hand, process algebras, CSP for example, supports process constructs such as parallel composition, deterministic and non-deterministic choices to describe system behaviours, although the data state and its operations cannot be concisely specified.

In order to solve the above problem, many specification languages integrating two or more existing languages have been proposed. Examples on enhancing CSP include *Circus* [95] (an integration of CSP and the Z language), CSP-OZ [23, 79] (an integration of CSP and Object-Z [80]) and TCOZ [46] (an integration of Timed CSP [74] and Object-Z). Because declarative specification languages like Z are very expressive and not executable, automated analysing (in particular, model checking) systems that are modelled using these integrated languages is extremely difficult. Another noticeable CSP extension is CSP_M [62] that is a machine-readable dialect of CSP. CSP_M combines CSP with a *functional* programming language. In addition, CSP_M is supported by two analysis tools, i.e., FDR (Failures Divergence Refinement) [45] and ProB [37], and has been applied to a number of systems [44, 70, 9].

Recently, $\text{CSP}\#$ [83] (short for Communicating Sequential Programs, pronounced as “CSP sharp”) has been proposed to model and analyse concurrent systems. It integrates high-level CSP-like process operators with low-level program constructs such as *assignments* and *while* loop. Shared variables are directly supported in $\text{CSP}\#$, and their operations are expressed as procedural codes; namely, operations are represented as terminating sequential programs,

which can be composed using the high-level (CSP-like) compositional operators. $\text{CSP}\#$ is supported by a general model checker PAT [82, 85] (short for Process Analysis Toolkit, available at <http://www.patroot.com>) for system analysis.

Although both $\text{CSP}\#$ and CSP_M support CSP-like modelling notations and can deal with similar types of concurrent systems, there are subtle and substantial differences between them. For example, concurrency is captured differently; CSP_M supports synchronous channel communications only, while $\text{CSP}\#$ supports both synchronous/asynchronous channels and shared variables. Those differences can lead to different verification capabilities empowered by their respective analysis tools, i.e., FDR and ProB for CSP_M , and PAT for $\text{CSP}\#$. Currently no comprehensive comparison of these CSP extensions is available, although such effort is important and desired for users to select appropriate languages/tools for various concurrent systems from the perspectives of modelling and verification.

In this thesis, we conduct a systematic comparison of $\text{CSP}\#$ and CSP_M . We firstly compare the syntactic differences in terms of data and process perspectives. To explore the deep difference of process operators, we next investigate the operational semantics of processes, and develop certain translation rules between these two languages. Last but not least, we compare their model checking techniques and measure their verification capabilities through eight benchmark systems. These benchmarks are designed from different perspectives, i.e., specification models involving puzzle solving and shared variables, and verification properties covering refinement checking and LTL checking.

Our above comparison focuses on the operational semantics of $\text{CSP}\#$ that interprets the behaviour of $\text{CSP}\#$ models using labelled transition systems (LTS). Nevertheless, the existing $\text{CSP}\#$ operational semantics [83] is not fully abstract. Two behaviourally equivalent processes with respect to the operational semantics may behave differently under some process context which involves shared variables, for instance. In other words, the operational semantics of $\text{CSP}\#$ is not compositional and thus lacks the support of compositional verification of

process behaviours. Meanwhile, model checking method based on the operational semantics is certainly limited by the state explosion problem. In practice, the method can only be used for checking the finite state transition systems. Therefore, there is a need for defining a compositional denotational semantics to explain the notations of the CSP# language and further developing theorem proving approach to complement the model checking approach for system verification, which are exactly the second and third piece work in this thesis.

Denotational semantics formalises the meaning of a language by constructing mathematical objects, called denotations, to represent the expressions in the language [94]. To define CSP# denotational semantics, we apply the Unifying Theories of Programming (UTP) [31], a unified framework for defining denotational semantics for programs across different programming paradigms, to constructing a semantic model which can cover not only communications but also shared variable paradigm. In our work, we firstly define an observation-oriented denotational semantics for CSP# based on the UTP framework in an *open* environment, where process behaviours can be interfered with by the environment. We also prove the compositionality properties of our proposed semantics for CSP# process combinators. Secondly, to make the semantics of CSP# more complete, on one hand, we construct the algebraic semantics by defining a set of algebraic laws [30] concerning the distinct features of CSP#. On the other hand, we define a *closed* semantics to capture restrictive behaviours where the model of a whole system has been built or the behaviour of the environment has been modelled.

Proving our deduced algebraic laws is important as such proofs can validate the correctness of our proposed CSP# denotational semantics. However, manual proving is tedious, and subtle mistakes or omissions can easily occur at any stage of the proofs. Moreover, a high grade of automated verification of system properties can save much human effort. Therefore, a tool that allows semantics mechanisation and supports mechanical proof is needed. In this thesis, we mechanise our defined denotational semantics into the Prototype Verification

System (PVS) [54], which is an integrated framework for formal specification and verification. Our mechanisation covers the semantic model, expressions, sequential programs, and $\text{CSP}\#$ processes, based on PVS built-in set theories. We use the predefined function `subset?` to represent the refinement relationship, and formalise the fixed point theory to represent recursive processes. In addition, we apply the PVS prover to validating the consistency of the semantics and mechanically proving essential laws of our formalisations so as to check the correctness of our encoding.

1.2 Thesis Outline

Our main work consists of the comprehensive comparison of $\text{CSP}\#$ and CSP_M in terms language syntax, operational semantics and reasoning power of their supporting tools, the construction of $\text{CSP}\#$ denotational semantics using the UTP framework, and the development of an interactive theorem proving framework for system verification.

Chapter 2 gives the background knowledge of specification languages and tools used in the presented work. We first review the $\text{CSP}\#$ specification language. Next we briefly describe the UTP theory with three essential elements. Lastly, we introduce the PVS specification language and its interactive prover.

Chapter 3 illustrates the comprehensive comparison of $\text{CSP}\#$ and CSP_M . Firstly, we show the syntactic differences of these two languages followed by comparing the operational semantics. We also discuss the possible transformation between $\text{CSP}\#$ and CSP_M models. Secondly, we characterise various reasoning techniques and verifiable properties of FDR, ProB and PAT, respectively. Next, we explore the strengths and limits of the languages and tools by modelling and verifying eight benchmark systems, each of which is designed to show particular features of the languages or the tools. Lastly, we investigate the reasons behind the experiment results; particularly, the semantic differences between $\text{CSP}\#$ and CSP_M lead

to different state spaces and optimizations in model checking.

Chapter 4 provides a UTP semantics for $\text{CSP}\#$. We firstly develop $\text{CSP}\#$ semantic model followed by detailed process semantics, based on the UTP framework in an open environment. Our semantic model includes observational variables which record process behaviours and healthiness conditions that identify the valid predicates of the theory for $\text{CSP}\#$. We also define refinement rules to check the equivalence of two $\text{CSP}\#$ specifications. Secondly, we present a set of algebraic laws that concerns the distinct features of $\text{CSP}\#$. All the laws are proved based on the denotational semantics. Lastly, we derive a closed semantics from the proposed open semantics by considering a closed environment.

Chapter 5 mechanises the denotational semantics of $\text{CSP}\#$ in PVS. First we formalise the definition of observational variables and healthiness conditions in PVS. Based on this formalisation, we encode the syntax and semantics of arithmetic expressions, Boolean expressions and sequential programs. Further, the semantics of processes and refinement relationship is mechanised. Lastly, we conduct mechanical proofs of important algebraic laws and lemmas based on the encoding in PVS.

Chapter 6 concludes the thesis with contributions and some possible future directions of research.

1.3 Acknowledgement of Published Work

Most of the work presented in this thesis has been published in international conference proceedings.

- **An Analytical and Experimental Comparison of CSP Extensions and Tools** [77].

This paper was published at the 14th International Conference on Formal Engineering Methods (ICFEM'2012). This work is presented in Chapter 3.

- **A UTP Semantics for Communicating Processes with Shared Variables [78].**

This paper was published at the 15th International Conference on Formal Engineering Methods (ICFEM'2013). This work is presented in Chapter 4.

In addition, the work in Chapter 5 has been submitted for publication.

We have published two papers which are relevant to my future work but not part of this thesis. The topics covered by these papers are:

- **Modeling and Verification of Transmission Protocols: A Case Study on CSMA/CD Protocol [76].** We conduct a case study on a transmission protocol, CSMA/CD protocol. We first model the protocol using the Stateful Timed CSP specification language, and then verify critical properties such as deadlock freeness and collision detection in a given bounded delay using the PAT tool.
- **Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP [84].** We propose a specification language called Stateful Timed CSP and an automated approach for verifying Stateful Timed CSP models. We further enhance the PAT model checker with the techniques to support the analysis of real-time systems and show its usability/scalability via verification of real-world systems.

Chapter 2

Background

In this chapter, we introduce some background information on the notations, theories and tools that are employed in this thesis. This chapter is divided into three parts. Section 2.1 introduces CSP# including its syntax and concurrency mechanisms. Section 2.2 gives a brief description of the UTP theory. The PVS specification language and interactive prover are presented in Section 2.3.

2.1 The CSP# Language

CSP# [83] integrates CSP-like compositional operators with sequential program constructs such as assignments and while loops. It directly supports shared variables which are not available in CSP [29]. Shared variables can be updated in sequential programs. Besides, CSP# is supported by a general model checker PAT [85].

2.1.1 Syntax

A CSP# model may consist of definitions of constants, variables, channels, and processes. A constant is defined by keyword *#define* followed by a name and a value, e.g., *#define max 5*. A global variable is declared with keyword *var* followed by a name and an initial value, e.g., *var x = 2*. A channel is declared using keyword *channel* with a name and a non-negative buffer size, e.g., *channel ch 2*. A process is specified in the form of $Proc(i_1, i_2, \dots, i_n) = ProcExp$, where *Proc* is the process name, (i_1, i_2, \dots, i_n) is an optional list of process parameters and *ProcExp* is a process expression. The BNF description of *ProcExp* is shown below with short descriptions, where *P* and *Q* are processes, *a* is an action, *e* is a non-communicating event, *ch* is a channel, *exp* is an arithmetic expression, *m* is a bounded variable, *prog* is a sequential program updating global shared variables¹, *b* is a Boolean expression, and X_1 is a set of actions.

$P ::= Stop \mid Skip$	– primitives
$a \rightarrow P$	– event prefixing
$ch!exp \rightarrow P \mid ch?m \rightarrow P(m)$	– channel output/input
$e\{prog\} \rightarrow P$	– data operation prefixing
$[b]P$	– state guard
$\text{if } (b) \{P\} \text{ else } \{Q\}$	– conditional choice
$P \square Q \mid P \sqcap Q \mid P \sqcup Q$	– external/internal/general choices
$P; Q$	– sequential composition
$P \setminus X_1$	– hiding
$P \parallel Q \mid P \parallel\!\!\mid Q$	– parallel/interleaving
$P \triangle Q$	– interrupt
$ref(Q)$	– process reference

In the above BNF description, process *Stop* is the process that communicates nothing and *Skip* is the process that terminates successfully. Event prefixing $a \rightarrow P$ engages in action *a* first and afterwards behaves as process *P*.

¹The grammar rules of the sequential program can be found in PAT user manual.

In CSP#, there are two types of channels, *synchronous* and *asynchronous*. A synchronous channel with buffer size 0 sends/receives messages synchronously and its communication is achieved by a pairwise handshaking mechanism. Specifically, a process $ch!exp \rightarrow P$ which is ready to perform an output through ch will be enabled if another process $ch?m \rightarrow P(m)$ is ready to perform an input through the same channel ch at the same time, and *vice versa*. An asynchronous channel with pre-defined buffer size sends/receives messages asynchronously. To be specific, process $ch!exp \rightarrow P$ evaluates the expression exp and puts the value of exp into the tail of the respective buffer and behaves as P ; process $ch?m \rightarrow P(m)$ gets the top element in the respective buffer, assigns it to variable m and then behaves as P .

In process $e\{prog\} \rightarrow P$, $prog$ is executed *atomically* with the occurrence of e . Process $[b]P$ waits until condition b becomes *true* and then behaves as P . Conditional choice **if** (b) $\{P\}$ **else** $\{Q\}$ behaves as P if b evaluates to *true*, and behaves as Q otherwise. There are three types of choices in CSP#: external choice $P \sqcap Q$ is resolved only by the occurrence of a *visible* event, internal choice $P \sqcap Q$ is resolved non-deterministically, and general choice $P \sqcup Q$ is resolved by any event.

Sequential composition $P; Q$ behaves as P until P terminates and then behaves as Q . Process $P \setminus X_1$ hides all occurrences of actions in X_1 . In process $P \parallel Q$, P and Q run in parallel, and they synchronise on common communication events. In contrast, in process $P \parallel\!\!\parallel Q$, P and Q run independently (except for communications through synchronous channels and shared variables). Process $P \triangle Q$ behaves as P until the first occurrence of a visible event from Q . A process expression may be given a name for referencing. Recursion is supported by process referencing.

2.1.2 Concurrency

As mentioned earlier, concurrent processes in CSP# can communicate through shared variables, events, or channels.

Shared variables in CSP# are globally accessible; they can be read and written by different (parallel) processes. Shared variables can be used in guard conditions, sequential programs associated with non-communicating events, and expressions in the channel outputs; nonetheless, they can only be updated in sequential programs. Furthermore, to avoid any possible data race problem when programs execute atomically, sequential programs from different processes are not allowed to execute simultaneously.

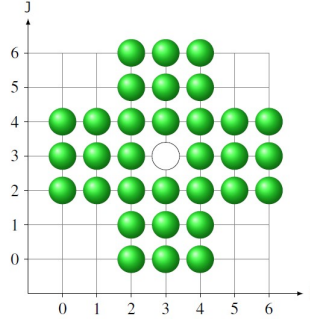
In CSP#, a synchronisation event, which is also called an action, occurs *instantaneously*, and its occurrence may require simultaneous participation by more than one processes. In contrast, a communication over a synchronous channel is two-way between a sender process and a receiver process. Namely, a handshake communication $ch.exp$ occurs when both processes $ch!exp \rightarrow P$ and $ch?m \rightarrow Q(m)$ are enabled simultaneously. We remark that this two-way synchronisation is different from CSP_M where multi-part synchronisation between many sender and receiver processes is allowed [62].

2.1.3 A CSP# Example - the Peg Solitaire Game

We here use the peg solitaire game as an example to elaborate the expressiveness of CSP#. This example is also one of the eight benchmark systems used in Chapter 3, Section 3.3.2.

Peg solitaire game is a game for one player to move pegs on a board with holes. A valid move is to jump a peg orthogonally over an adjacent peg into a hole which is two positions away and then remove the jumped peg. There are four orthogonal directions, namely, up, down, left, and right. The goal is to empty the entire board except for a solitary peg in the

initial empty hole. The following picture shows the starting of this game with 32 pegs.



We first model the board as a global shared variable which is a two-dimension array.

```
var board[H][W] =
  [X, X, P, P, P, X, X,
   X, X, P, P, P, X, X,
   S, P, P, P, P, P, P,
   P, P, P, E, P, P, P,
   P, P, P, P, P, P, P,
   X, X, P, P, P, X, X,
   X, X, P, P, P, X, X];
```

Here H and W are the size of the row and column for the board, X is a non-slot, P is a slot with a peg and E is an empty slot. All of them are defined as constant integers.

```
#define X -1; #define P 1; #define E 2; #define W 7; #define H 7;
```

Next we capture four directions of jumps of a peg. Each jump is represented by a process. For example, process $Up(i, j)$ denotes that a peg at the position (i, j) jumps upwards, provided its upside position is occupied by a peg and the destination is empty (represented by guard $[board[i-2][j] == E \ \&\& \ board[i-1][j] == P]$); after the jump, the original and the jumped positions become empty (by the sequential program $board[i-2][j] = P; board[i-1][j] = E; board[i][j] = E$). Here variable *pegsCounter* indicates the number of existing pegs on the board; initially its value is 32, and the value is reduced by 1 after a valid move (by

$pegsCounter - -$). Process $Peg(i, j)$ uses external choice to model all possible jumps of a peg.

```

var pegsCounter = 32;
Up(i, j) = [i - 2 >= 0]([board[i - 2][j] == E && board[i - 1][j] == P]
  up{board[i - 2][j] = P; board[i - 1][j] = E; board[i][j] = E;
    pegsCounter - -; } → Game());
Left(i, j) = [j - 2 >= 0]([board[i][j - 2] == E && board[i][j - 1] == P]
  left{board[i][j - 2] = P; board[i][j - 1] = E; board[i][j] = E;
    pegsCounter - -; } → Game());
Down(i, j) = [i + 2 < H]([board[i + 2][j] == E && board[i + 1][j] == P]
  down{board[i + 2][j] = P; board[i + 1][j] = E; board[i][j] = E;
    pegsCounter - -; } → Game());
Right(i, j) = [j + 2 < W]([board[i][j + 2] == E && board[i][j + 1] == P]
  right{board[i][j + 2] = P; board[i][j + 1] = E; board[i][j] = E;
    pegsCounter - -; } → Game());
Peg(i, j) = [board[i][j] == P](Up(i, j) □ Left(i, j) □ Down(i, j) □ Right(i, j));

```

To complete our model, process $Game$ covers the behaviour of any peg on the board using the external choice.

```

Game() = □ i : {0..H - 1}; j : {0..W - 1}@ Peg(i, j);

```

We can leverage the PAT model checker to deduce a solution by checking a reachability assertion, where the goal condition specifies that only one peg is on the board at the initial empty hole.

```

#define initEmptyX 3; #define initEmptyY 3;
#define goal pegsCounter == 1 && board[initEmptyX][initEmptyY] == P;
#define assert Game() reaches goal;

```

2.2 UTP Theory

The Unifying Theories of Programming (UTP) [31] is proposed by Hoare and He, to deal with program semantics. It is a unified framework to combine denotational semantics, operational semantics, and algebraic semantics for formal specification, design and implementation of programs and computer systems. UTP uses the theory of relations as a unifying basis to define denotational semantics for programs across different programming paradigms, e.g., imperative programming paradigm (C language for instance), functional programming paradigm (such as LISP [47], ML [50, 89], and Haskell [33]), and high order programming paradigm (e.g., Java language).

For each programming paradigm, programs are generally interpreted as relations between initial observations and subsequent (intermediate or final) observations of the behaviours of their execution. Relations are represented as predicates over observational variables to capture all aspects of program behaviours.

Theories of programming paradigms in the UTP framework are differentiated by their *alphabet*, *signature* and *healthiness conditions*. The alphabet is a set of observational variables recording external observations of the program behaviour. The signature defines the syntax to represent the elements of a theory. The healthiness conditions are a selection of laws identifying valid predicates that characterise a theory.

The observational variables in the alphabet of a theory record the observations that are relevant to program behaviours. Variables of initial observations are undashed, constituting the input alphabet of a relation, and variables of subsequent observations are dashed, constituting the output alphabet of a relation. For example, in the imperative paradigm, variables x, y, \dots, z record the initial state of program variables, and x', y', \dots, z' record the final state of program variables. In a theory of reactive processes, Boolean variable *wait* distinguishes the intermediate observations of a waiting state from the observations of a

final state for reactive processes; Boolean variable *ok* records the stability of program, i.e., whether it is in a stable state or in a divergent state; variable *tr* records the interaction between a process and its environment; *ref* records the set of events that could be refused before the observation.

The signature of a theory is a set of atomic components called *primitives* and *combinators*. The primitives in the signature of relational programming are assignment $x := e$, empty *skip*, top \top for miracle and bottom \perp for abort. The combinators are conditional $P \triangleleft b \triangleright Q$, composition $P; Q$, nondeterminism $P \sqcap Q$ and recursion $\mu X \bullet F(X)$. Here, x is a variable in the alphabet, e is an expression, P and Q are predicates describing behaviours of two programs, X is a recursive variable standing for a predicate, and F is a monotonic function.

A healthiness condition is associated with observational variables in the alphabet. It is defined by an idempotent function ϕ on predicates. Every healthy program represented by predicate P must be a fixed point: $P = \phi(P)$.

For example, if a program has not started, the observation of its behaviour is impossible. This can be captured by a healthiness condition $H(P) = ok \Rightarrow P$ requiring that program satisfies the following equation:

$$P = H(P) \text{ or } P = ok \Rightarrow P.$$

In the above example, if Boolean variable *ok* is *true*, then program starts and its behaviour is described by predicate P . If *ok* is *false*, then its behaviour is not restricted as predicate $ok \Rightarrow P$ is *true*.

2.3 Prototype Verification System

Prototype Verification System (PVS) [54, 19] is an integrated environment for the development and analysis of formal specifications. It combines an expressive modelling language with an interactive prover that has powerful theorem proving capabilities.

The specification language is based on classical typed higher-order logic. Its type system consists of base types such as *Boolean* (`bool`), *integer* (`int`), *real numbers* (`real`) and type constructors for *function types*, *tuple types*, and *record types*. A function type is usually of the form $[D \rightarrow R]$, where D and R are type expressions, denoting the domain and range of the function respectively. Tuple types (also called product types) have the form $[T_1, \dots, T_n]$, where the T_i are type expressions. Projection function ‘ i ’ is used to project the i th element of the tuple. Record types are of the form $[# a_1:T_1, \dots, a_n:T_n \#]$. The a_i are called record accessor or fields and the T_i are types. For example, a record type R consisting of an integer number x and a Boolean variable b is specified as $R:TYPE = [# x: int, b: bool]$, given a record $r: VAR R$, its x -component is accessed by $r.x$.

The type system of the PVS is augmented with *predicate subtypes* and *dependent types*. Subtypes can be specified in two different ways. Given a type X and predicate P on the elements of X , a subtype of X with respect to P can be specified as either $T: TYPE = \{x:X | P(x)\}$ or $T: TYPE = (P)$. The type checking of subtypes is undecidable, and may lead to proof obligations, called *type correctness conditions* (TCCs). Users are required to discharge these TCCs with the assistance of the PVS prover.

Another important feature of PVS type system is the provision of abstract datatypes. Familiar data structures of programming languages such as lists and binary trees can be specified in PVS using the abstract datatypes. For example, the following PVS specification declares a list using abstract datatype.

```

list [T: TYPE]: DATATYPE
  BEGIN
    null: null?
    cons (car: T, cdr:list):cons?
  END list

```

To be specific, `list` is parametric in type `T`, and has two constructors `null` and `cons`: `null` takes no arguments and `cons` takes two arguments, where the first is of the type `T` and the second is a list. Two predicates `null?` and `cons?` are recognisers: `null?` holds for exactly those elements of the `list` datatype that are identical to `null`, and `cons?` holds for exactly those elements of the `list` datatype that are constructed using `cons`. Note that two accessors, `car` and `cdr`, correspond to the two arguments of `cons`; they can only be applied to lists which satisfy the `cons?` predicate.

A PVS specification is given as a collection of parameterised *theories*. Each theory may consist of declarations, definitions and formulas. Declarations are used to define types, variables, constants, and so on. *Type declarations* introduce new type names to the theory. *Variable declarations* introduce new variables with their associated types. In addition, variables are local when they are defined in binding expressions which may involve keywords such as `FORALL` for the universal quantifier \forall and `LAMBDA` for the symbol λ in lambda expressions. *Constant declarations* introduce new constants with their associated types and an optional value, and constants can be functions, relations or the usual (0-ary) constants. For example, the declaration `f: [nat -> nat] = (lambda (x: nat): x + 1)` defines a total function `f` (by the symbol `->`) where, the domain and range are natural numbers, and its output value is one more than input value.

PVS supports *recursive definitions*, which are total functions. Hence, it must be ensured that all recursive functions terminate, specified by a *measure* expression. The measure expression follows the `MEASURE` keyword and ends with an optional order relation following a `BY` keyword. The recursive definition generates a *termination TCC* which denotes that

the measure function applied to recursive arguments decreases with respect to a well-formed ordering. A proof obligation must be discharged by users.

Formula can be declared to introduce axioms using the keyword **AXIOM** and theorems using the keyword **LEMMA**. Axioms can be referenced by the command **lemma** during proofs. The body of the formula is a Boolean expression. Moreover, PVS supports the *name overloading* technique which allows the same name from different theories or within a single theory. The collections of theories are organised by means of importings.

The PVS prover [75] is based on a sequent calculus and proofs are constructed interactively by building a *proof tree*. The goal of users is to construct a complete proof tree where all of the leaves are recognised true. Each node in a proof tree is a *proof goal* which is a sequent consisting of a list of formulas called *antecedents* and a list of formulas called *consequents*. The intuitive interpretation of a proof goal is that the conjunction of the antecedents implies the disjunction of the consequents.

The PVS prover provides a collection of powerful proof commands to perform induction, propositional and equality reasoning, rewriting, model checking and so on. For example, a frequently used proof command is **grind**, which does skolemization, instantiation, simplification, rewriting and applying decision procedures.

Chapter 3

Comparison of CSP Extensions and Tools

Communicating Sequential Processes (CSP), a prominent member of the process algebra family, has been designed to formally model *concurrent systems*. It has been applied to a variety of safety-critical systems [96, 7, 27]. With the increasing size and complexity of concurrent systems, CSP becomes deficient to model systems with non-trivial data structures (e.g., *array*) or functional aspects. To solve this problem, many considerable efforts on enhancing CSP have been made. Two noticeable extensions with automated tool support are CSP_M and $\text{CSP}\#$. The former combines CSP with a *functional* programming language, and the latter integrates CSP-like process operators with *sequential* programs. Although these CSP-like extensions support similar types of concurrent systems, subtle and substantial differences exist including not only concurrency mechanisms provided by the languages, but also verification capabilities empowered by their analysis tools.

A comprehensive investigation of these CSP extensions from various perspectives would certainly facilitate users to determine appropriate languages/tools for concurrent systems

with specific characteristics. The above importance motivates us to conduct a systematic and thorough comparison between CSP_M and $CSP\#$ as the first attempt to our best knowledge.

Our comparison can benefit users from the following three aspects. First, assessment criteria are proposed for choosing a suitable modelling language, which includes special system features such as shared variables and desired properties like compositional refinement. Next, our experiments with eight benchmark systems offer in-depth qualitative analysis of tool capability and efficiency, specifically, FDR [45] and ProB [37] for CSP_M and PAT [85] for $CSP\#$. Last but not least, transformation discussed in this chapter can help users to change their models between CSP_M and $CSP\#$, and hence to utilize different reasoning power of their respective reasoning tools.

This remainder of the chapter is organized as follows. Section 3.1 shows syntactic differences between CSP_M and $CSP\#$ in terms of data and process perspectives. Section 3.2 compares the operational semantics of CSP_M and $CSP\#$. Section 3.3 investigates the properties and verification techniques of FDR, ProB and PAT with experiments on eight benchmark systems. Section 3.4 concludes this chapter with the discussion of related work.

3.1 CSP_M vs. $CSP\#$: Syntax

CSP_M enriches CSP with an expression language that is based on *functional* foundations. It mainly uses event synchronisation to specify concurrent systems, and supports operators like linked parallel $P[c < - > c']Q$ in which two different channels c and c' from processes P and Q respectively run synchronously. $CSP\#$ not only inherits event synchronisation and compositional process constructs from CSP, but also supports additional features like asynchronous channel communication, imperative programs, etc.

In this section, we elaborate the differences between these two languages in terms of their

syntax. Table 3.1 shows common process definitions of CSP, CSP_M and $CSP\#$ ¹, where P (and Q) is a process with an optional list of parameters; a is an event name; A and A' are sets of event names and channel expressions; b is a Boolean expression; c and c' are channel names; e is an expression; x and x' are variables; and V is a set of accepted values. We illustrate the detailed differences from data and process perspectives, shown in Sections 3.1.1 and 3.1.2 respectively.

CSP	CSP_M	$CSP\#$	Description
$STOP$	$STOP$	$Stop$	deadlock
$SKIP$	$SKIP$	$Skip$	termination
$CHAOS$	$CHAOS(A)$	-	chaotic process
$a \rightarrow P$	$a \rightarrow P$	$a \rightarrow P$	event prefixing
$c!e \rightarrow P$ $c?x \rightarrow P$	$c?x?x' : V!e \rightarrow P$	$c!e \rightarrow P$ $c?[b]x \rightarrow P$	channel communication
$P \square Q$	$P \sqcup Q$	$P [*] Q$	external choice
$P \sqcap Q$	$P \sim Q$	$P <> Q$	internal choice
$P; Q$	$P; Q$	$P; Q$	sequential composition
$P \setminus A$	$P \setminus A$	$P \setminus A$	hiding
$x := e$	-	$x := e$	assignment
$P \triangleleft b \triangleright Q$	$if\ b\ then\ P\ else\ Q$	$if\ b\ then\ P\ else\ Q$	conditional choice
$P \parallel Q$	$P[A]Q$ $P[A A']Q$ $P[c < - > c']Q$	$P \parallel Q$	parallel composition
$P Q$	$P Q$	$P Q$	interleaving
$P \triangle Q$	$P / \setminus Q$	$P\ interrupt\ Q$	interrupt

Table 3.1: Similar syntax among CSP, CSP_M and $CSP\#$

3.1.1 Data Perspective

CSP_M supports functional paradigm, where higher-order functions and declarative programming are supported. In addition, process parameters in CSP_M can be processes, functions, and channels. This is not available in $CSP\#$ which adopts imperative paradigm, although this limitation may be resolved partially through ‘clever’ modelling. For in-

¹Note that these are $CSP\#$ representations in PAT text editor.

stance, a CSP_M concrete process $System = P(Sys1, Sys2)$ associated with an abstract process $P(P1, P2) = a \rightarrow P1 \parallel b \rightarrow P2$ can be translated to a $CSP\#$ concrete process $System = a \rightarrow Sys1 [*] b \rightarrow Sys2$, where $Sys1$ and $Sys2$ are processes. However, it may not be possible to specify abstract process behaviour (e.g., process P in this example) in $CSP\#$, whose parameters are processes.

CSP_M enables rich data expressions such as sequences, sets, Boolean, tuples, and lambda calculus. It also allows users to define data types using the reserved word “datatype”. $CSP\#$ directly supports integers, Boolean, array of integers or Boolean. In addition, it supports user-defined data types and corresponding operations using imperative languages like C^{#2}, C, or Java. Functions can be declared in CSP_M following the functional paradigm, while in $CSP\#$, they are encoded as processes or defined as static C[#] methods (which can be invoked via method *call* in $CSP\#$ models).

A channel in CSP_M is declared with an explicit type. Values communicated through a channel must be in their type range; otherwise, an error is reported at run time by FDR and ProB. Moreover, CSP_M is dynamically typed in FDR; namely, there is no way to declare the types of functions and variables (process parameters), while ProB can type check the CSP_M models in a dynamic or (optional) static way [38]. In contrast, $CSP\#$ is weak typed (a.k.a. loose typing) and therefore no type information is required when declaring a variable or channel. Channels are declared with its name and buffer size. If the buffer size is 0, then it is declared as a synchronous channel, otherwise it is an asynchronous channel. The process parameters and channel input variables can take in values with different types at different time. As long as there is no type mismatch (e.g., using an integer as a guard condition), the execution can proceed; otherwise, invalid type casting exception is raised at run time.

²C[#] is the best supported language in PAT and used as the representative language in this chapter.

3.1.2 Process Perspective

One big difference is that $CSP\#$ directly supports shared variables. Unlike CSP_M which excludes assignments of shared variables [45], $CSP\#$ treats assignments as an important modelling feature. In $CSP\#$, an event can be associated with an imperative program, which is executed *atomically* together with the occurrence of the event. For instance, an event associated with a program (referred to as a data operation) is written as $a\{prog\} \rightarrow P$ where $prog$ is the program and a is an event name. We remark that a shared variable can be modelled as a process parallel to the one that uses the variable (see [29] and [66]). Recently, shared variable analyser (SVA) [66], a front-end of FDR, has been developed to convert programs (like C programs) with shared variables into CSP_M models, in which shared variables are modelled as *variable processes*; reading from/writing to those shared variables are carried out over channels. We illustrate the modelling of shared variables in Section 3.2.4.

Asynchronous channels, as a popular and practical type of communication mechanism for networked systems, are directly supported in $CSP\#$. Given an asynchronous channel ac with a positive buffer size, $ac!e \rightarrow P$ evaluates expression e with the current variable valuation, puts the value into the tail of the respective buffer for ac and then behaves as P . In contrast, $ac?x \rightarrow P$ (and $ac?[b]x \rightarrow P$) gets the top element from the respective buffer, assigns it to variable x and then behaves as P (the latter further constrains the received data to satisfy the Boolean condition b). Buffers store messages in a first-in-first-out (FIFO) order. Notice that asynchronous channels in $CSP\#$ are similar to those supported in Promela [32]. Although asynchronous channels are not directly supported in CSP_M , they can be modelled as buffer processes by event synchronisation, which will be shown in Section 3.2.3.

In CSP_M , users are required to indicate synchronised events in three kinds of parallel compositions, which are, *sharing* ($P \parallel A \parallel Q$) that runs processes P and Q in parallel and forces

them to synchronise on events in set A , *alphabetized parallel* $(P[A \parallel A']Q)$ that runs P and Q in parallel, allowing P and Q to only perform events from A and A' respectively, and forcing P and Q to synchronise on common events from A and A' , and *linked parallel* $(P[c \leftrightarrow c']Q)$ that runs P and Q in parallel, forcing them to synchronise on the c and c' events and then hides the synchronised events. On the other hand, $CSP\#$ supports only alphabetized parallel composition and frees users from specifying explicit alphabets of processes in parallel; a sophisticated procedure [83] calculates automatically a *default* alphabet of a process which is the set of events that constitute the process expression. Nevertheless, this procedure may not work when an event name consists of global variables or process parameters which change through *recursive calls*; in such a case, users need to specify the alphabet of a process. Notice that in order to avoid data race, data operations are not a part of the alphabet and therefore are never synchronised.

In $CSP\#$, an event can have the name *tau* to represent the invisible event τ in event prefixing or data operations, e.g., $\tau \rightarrow Stop$ or $\tau\{prog\} \rightarrow Stop$. With the support of *tau* event, users can avoid using hiding operator to explicitly hide some visible events by naming them *tau*. *External* and *internal* choices are supported in both languages. Moreover, $CSP\#$ allows *general choice* $P \square Q$ in which the choice is resolved by any event. This operator is more like the CCS $+$ operator, which can be resolved by a τ event performed by either process. Nonetheless, the general choice operator can be simulated in CSP_M [65].

Besides the above common conditional choice, $CSP\#$ copes with two additional types of conditional choices to facilitate modelling: *atomic* conditional choice *ifa* $b \{P\} \text{ else } \{Q\}$ and *blocking* conditional choice *ifb* $b \{P\}$. With the former, the checking of condition b is to be conducted *atomically* with the occurrence of the first event in P or Q . The latter is blocked when b is unsatisfied.

Both CSP_M and $CSP\#$ define Boolean guard $b \& P$ and $[b]P$ respectively; a process waits until condition b becomes true and then behaves as P . Replicated process operators, such as

replicated external/internal choices, replicated parallel and interleaving, are also supported in both languages. Chaotic process ($CHAOS(A)$), event renaming ($P[[c \leftarrow c']]$), and untimed timeout ($P[> Q]$) defined in CSP_M are not directly handled in $CSP\#$. We discuss how to model these features using $CSP\#$ operators in Section 3.2.

So far we have shown the syntactic differences between CSP_M and $CSP\#$. Both CSP_M and $CSP\#$ support dedicated syntax which is unavailable in the other. Some special syntax operators in one can be indirectly achieved in the other. For instance, the *CHAOS* process in CSP_M can be defined in $CSP\#$ using choices and event prefixing (discussed in the next section). Nonetheless, it is not always trivial to support some dedicated syntax operators such as shared variables in CSP_M and channel communications in $CSP\#$ (which can involve multiple processes).

3.2 CSP_M vs. $CSP\#$: Operational Semantics

Operational semantics describes the sequences of computational steps that a model can take. We illustrate the operational semantics of CSP_M and $CSP\#$ in the form of labelled transition systems (LTS). An LTS is a tuple $\mathcal{L} = (S, init, \rightarrow)$ where S is a set of system configurations; $init \in S$ is an initial system configuration and $\rightarrow: S \times \Sigma \cup \{\checkmark, \tau\} \times S$ is a labelled transition relation. Note that $\Sigma \cup \{\checkmark, \tau\}$ is the event space where Σ is the set of *visible* events, \checkmark denotes a successful termination, and τ is an *invisible* event.

A system configuration S in CSP_M is a *pair* of processes and environment where the latter maps variable identifiers to values such as data, processes, or a distinguished *error* configuration. In $CSP\#$, S is composed of two components (V, P) where V maps variable names (or channel names) to values (or sequences of items in buffers), and P is a process expression. The operational semantics of a process construct is depicted by associated firing rule(s). CSP_M and $CSP\#$ share similar firing rules for some process constructs like

interrupt [62, 72, 83]. In the following subsections, we elaborate the differences in terms of operational semantics of all process constructs. Note that the firing rules for CSP_M adopt a simple approach which only deals with processes with no free identifiers, same as [62], thus environment in CSP_M is not included in the firing rules below.

3.2.1 SKIP

Process *SKIP* means termination; namely, \checkmark takes place followed by doing nothing, as captured by *Stop* in $CSP\#$, whereas this is denoted by a special process term Ω in CSP_M . For simplicity, we use prefix M to refer to CSP_M firing rules (e.g., M_skip), and $\#$ for $CSP\#$ (e.g., $\#_skip$) below.

$$\frac{}{SKIP \xrightarrow{\checkmark} \Omega} \quad [M_skip] \qquad \frac{}{(V, Skip) \xrightarrow{\checkmark} (V, Stop)} \quad [\#_skip]$$

Notice that in both CSP_M and $CSP\#$, \checkmark may only be the last event of a trace. The semantic difference shown above thus will not result in different verification results in FDR, ProB and PAT³. Nonetheless, it should be noticed that this difference leads to a different semantics for parallel composition as we show later.

3.2.2 CHAOS

Process *CHAOS* in CSP_M denotes the most non-deterministic process.

$$\frac{}{CHAOS(A) \xrightarrow{\tau} STOP} \quad [M_c1] \qquad \frac{a \in A}{CHAOS(A) \xrightarrow{a} CHAOS(A)} \quad [M_c2]$$

³except deadlock-freeness checking; namely, a process is deadlock free *iff* it satisfies the deadlock-freeness assertion in FDR and ProB, whereas it has to satisfy both deadlock-freeness and nontermination assertions in PAT.

$CHAOS(A)$ is not directly supported by $CSP\#$ because of two main reasons. First, users have to specify all the events in set A to model $CHAOS$, whereas $CSP\#$ is designed to free users from specifying events associated with processes (if possible). Second, $CHAOS$ is more useful in the failures/divergences checking, whereas $CSP\#$ models focus more on states/LTL checking. $CHAOS(A)$ can be manually captured in $CSP\#$ by constructing an equivalent process including all events. For example, let set A contains events a and b , one way to model $CHAOS(A)$ process in $CSP\#$ can be as follows.

$$CHAOS_A = \tau \rightarrow Stop \sqcup a \rightarrow CHAOS_A \sqcup b \rightarrow CHAOS_A$$

3.2.3 Channel Communication

Channel communications are crucial in concurrent systems and they are classified into two types: *synchronous* and *asynchronous*. CSP_M directly supports the former, whereas $CSP\#$ supports both. Both languages have their own operational semantics to interpret channel communications, which is elaborated below. The transformation of channel communication between CSP_M and $CSP\#$ is discussed later.

A general format to express a channel communication is $cf \rightarrow P$, where c is a channel name, f a sequence of communication fields, and P a process with the scope of the prefix. A communication field can be an output (by $!e$ where e is an expression), an unconstrained input (by $?x$ where x is a variable), or a constrained input (by $?x : V$ in CSP_M where V is a value range, and by $?[b]x$ in $CSP\#$ where b is a Boolean condition).

In CSP_M , channels are synchronous and communications are achieved by means of event synchronisation. Specifically, assuming the type of data communicated over channel c is T , $c!e \rightarrow P$ outputs a communication $c.v$ where v is the value of e and $v \in T$, and $c?x \rightarrow P$ accepts an input of the form $\{c.v \mid v \in T\}$; $c?x : V \rightarrow P$ imposes an additional constraint for $c.v$, namely, $v \in V$. As a channel can be associated with a sequence of

communication fields in CSP_M , multi-part communications involving multiple data transfers can occur within a single action. For instance, $c?x : V!e \rightarrow P$ engages communications of the form $\{c.v'.v \mid v'.v \in T \wedge v' \in V\}$ where v is a value of e . The firing rule of the CSP_M channel communication is presented below, where function $comms(cf)$ returns the set of communications described by cf and function $subs(a, cf, P)$ returns a process whose identifier in process P bounded by cf is substituted by event a .

$$\frac{a \in comms(cf)}{cf \rightarrow P \xrightarrow{a} subs(a, cf, P)} \quad [M_com]$$

In $CSP\#$, a channel is defined as a buffer which stores messages in a first-in-first-out (FIFO) order. Channels are synchronous when their buffer sizes are zero, in which case communications are realized by the hand shaking mechanism. Channels are asynchronous when their buffer sizes are bigger than zero, and their communications are achieved by the message passing mechanism. Sending and receiving multiple messages at one time are supported in both synchronous and asynchronous communications. We show below the firing rules of $CSP\#$ for channel communications.

- A synchronous communication occurs when both processes $c!e \rightarrow P$ and $c?x \rightarrow P$ (or $c?[b]x \rightarrow P$) can be executed *simultaneously* and the messages passed match (and condition b is true); event $c.v$ is transferred where v is the value of e with the *latest* valuation $eva(V, e)$. In the following firing rule which is associated with parallel composition (the case for interleaving is similar), process $Q[eva(V, e)/x]$ replaces x with the new value v .

$$\frac{\begin{array}{l} (V, c!e \rightarrow P) \xrightarrow{c!eva(V, e)} (V, P), (V, c?[b]x \rightarrow Q) \xrightarrow{c?[b]x} (V, Q), \\ (V \wedge x = eva(V, e)) \Rightarrow b \end{array}}{(V, c!e \rightarrow P \parallel c?[b]x \rightarrow Q) \xrightarrow{c.eva(V, e)} (V, P \parallel Q[eva(V, e)/x])} \quad [\#_par1]$$

- An output process $ac!e \rightarrow P$, where ac is an asynchronous channel, is enabled if the

associated buffer is not full. The process first evaluates e and then pushes the value into the tail of respective buffer for ac (denoted by function $app(V, ac!e)$), followed by the execution of P .

$$\frac{ac \text{ is not full in } V}{(V, ac!e \rightarrow P) \xrightarrow{ac!eva(V, e)} (app(V, ac!e), P)} \quad [\#_{out}]$$

- A constrained input process $ac?[b]x \rightarrow P$ is enabled if the associated buffer size is not empty and b is valid with the latest valuation (denoted by function $top(ac)$). The process pops (denoted by function $pop(V, ac?x)$) and assigns the top element from the buffer to x , followed by the execution of P . Note that the checking of b is unnecessary for an unconstrained input process.

$$\frac{ac \text{ is not empty in } V \wedge (V \wedge x = top(ac)) \Rightarrow b}{(V, ac?[b]x \rightarrow P) \xrightarrow{ac?top(ac)} (pop(V, ac?x), P[top(ac)/x])} \quad [\#_{in}]$$

We exemplify below how $CSP\#$ captures CSP_M multi-part synchronous channels and how $CSP\#$ asynchronous channels are represented in CSP_M . The event-like channel communication in CSP_M can be modelled as alphabetised event-based synchronisation in $CSP\#$. We capture the channel communication by expanding the channel values according the type values. Specifically, an output process $c!e \rightarrow P$ is translated to a process $c.e \rightarrow P$ in $CSP\#$, and an input process is transformed into a $CSP\#$ model which enumerates *all* possible communications using the general choices (\parallel) to combine relevant event prefixing processes. Taking the following CSP_M model of a vending machine (VM) as an example,

1. *datatype* $Drink = Sprite \mid Coke \mid Tea \mid Coffee$
2. *channel* $offer : Drink$
3. $VM = offer?x : diff(Drink, \{Coffee\}) \rightarrow VM$

where process VM can perform any communication in the form $\{offer.x \mid x \in diff(Drink, \{Coffee\}) \wedge x \in Drink\}$; function $diff(Drink, \{Coffee\})$ restricts that a vending machine can offer any drink except coffee. This VM can be captured by the following $CSP\#$ process

where all possible communications are explicitly specified.

$$VM = offer.Sprite \rightarrow VM \parallel offer.Coke \rightarrow VM \parallel offer.Tea \rightarrow VM$$

An asynchronous channel in $CSP\#$ can be modelled as a CSP_M process which represents the FIFO buffer by sending/receiving messages to/from other processes. We provide such a CSP_M process below, where a sequence is defined in process *Buffer* to store the message in the FIFO order, and *rcv* and *snd* are channels.

1. $Buffer(c, \langle \rangle, N) = rcv?c?x \rightarrow Buffer(c, \langle x \rangle, N)$
2. $Buffer(c, s \cap \langle a \rangle, N) = \#s < N - 1 \& rcv?c?x \rightarrow Buffer(c, \langle x \rangle \cap s \cap \langle a \rangle, N)$
 $\parallel snd!c!a \rightarrow Buffer(c, s, N)$

In the above *Buffer* process, line 1 describes the situation where the buffer is empty, and hence only receiving messages from other process is allowed. Line 2 depicts message receiving and sending when the buffer is not full. This *Buffer* process can be used to run in parallel with other process, say *P*, to perform asynchronous channel communication; for instance, a communication over an asynchronous channel *ac* with buffer size 2 can be modelled as $P[snd \leftrightarrow rcv, rcv \leftrightarrow snd]Buffer(ac, \langle \rangle, 2)$. We remark that asynchronous channel can be regarded as a special kind of shared variable, which is discussed in the next section; the above way modelling asynchronous channels in CSP_M is similar to the way of handling shared variables in CSP_M .

3.2.4 Shared variables

Shared variables are important in modelling shared resources. Variables in Hoare's CSP processes are local and disjoint. We elaborate below how shared variables are supported by $CSP\#$ directly and CSP_M indirectly.

$CSP\#$ uses shared variables to model data states and operations in a procedural style.

The operations are modelled as terminating sequential programs in the form $a\{prog\} \rightarrow P$, where programs $prog$ can contain local variables⁴, if-then-else statements, while loops, the invocation of external libraries written in C#/Java (through the *reflection* techniques). The execution of the programs is atomic together with the occurrence of associated events. In the following firing rules, function $upd(V, prog)$ returns a modified valuation function according to the particular semantics of the program; in $prog$, both shared and local variables can be used and updated.

$$\frac{}{(V, a\{prog\}^5 \rightarrow P) \xrightarrow{a} (upd(V, prog), P)} \quad [\#_dataOp]$$

Shared variables can be modelled in CSP_M indirectly as discussed in [66]. To be specific, a shared variable is represented by a *variable process* which is executed concurrently with other *user processes* which invoke the variable. Variable processes are modelled as read/write operations, and hence user processes can read from/write to the shared variables by CSP_M synchronous communication. For example, the following processes $Var(v, val)$ and $Var_A(j, v, val)$ execute together as a variable process to denote a shared variable v , where val is the value of v and j denotes a unique id of a user process which invokes v . The constraint that only one process is allowed to read/write v is specified in Var_A which is triggered by event $start_at?j!v$ from Var .

1. $Var(v, val) = read?i!v!val \rightarrow Var(v, val)$
2. $\square write?i!v?x \rightarrow Var(v, x) \square start_at?j!v \rightarrow Var_A(j, v, val)$
3. $Var_A(j, v, val) = read.j!v!val \rightarrow Var_A(j, v, val)$
4. $\square write.j!v?x \rightarrow Var_A(j, v, x) \square end_at?j!v \rightarrow Var(v, val)$

The following $CSP_\#$ model and CSP_M model represent the same system which sums three process parameters, where the processes are selected non-deterministically from three processes. In the $CSP_\#$ model, sum and $count$ are shared variables with initial value 0, and

⁴The scope of local variables is within $prog$, and they are not stored in valuation function V .

⁵Event a can also be an invisible event, denoted as tau , then the transition event becomes τ .

their updates are executed atomically with the occurrence of event *add* in process $P(i)$.

1. $var\ count = 0; \quad var\ sum = 0;$
2. $P(i) = [count < 3]add\{sum = sum + i; \quad count = count + 1; \} \rightarrow P(i);$
3. $System() = ||| \quad i : \{1..3\}@P(i);$

In the CSP_M model below, the shared variables *sum* and *count* are modelled as variable processes $Var(sum, 0)$ and $Var(count, 0)$. In addition, process $P(i)$ is defined by a sequence of variable access events (e.g., events *start_at!i!count* and *end_at!i!count* for *count*).

1. $datatype\ VarDt = count \mid sum \quad T = \{1..3\} \quad Range = \{0..10\}$
2. $P(i) = start_at!i!count \rightarrow read!i?count?x \rightarrow x < 3 \ \& \ add$
 $\rightarrow start_at!i!sum \rightarrow read!i?sum?y \rightarrow write!i!sum!(y + i)$
 $\rightarrow write!i!count!(x + 1) \rightarrow end_at!i!sum \rightarrow end_at!i!count \rightarrow P(i)$
3. $Processes() = ||| \quad i : \{1..3\}@P(i)$
4. $Variables() = Var(count, 0) \mid Var(sum, 0)$
5. $SharedEvent = \{read.t.v.val, \quad write.t.v.val, \quad start_at.t.v, \quad end_at.t.v \mid$
 $t \leftarrow T, \quad v \leftarrow VarDt, \quad val \leftarrow Range\}$
6. $System() = Variables() \mid SharedEvent \mid Processes()$

As shown above, $CSP\#$ allows users to specify shared variables and their operations in a way similar to imperative programming languages, which allows users to see variable states at each simulation step. In contrast, CSP_M supports shared variables by means of auxiliary processes and events; the additional operations may result in more system states during model checking, as shown later in our experiments.

3.2.5 Parallel composition

The firing rules of parallel composition $P \parallel Q$ in CSP_M and $CSP\#$ are similar except the way of handling the \checkmark event. Both languages require *distributed termination*: process $P \parallel Q$ terminates if both P and Q terminate. This requirement is satisfied in $CSP\#$ by the following firing rule.

$$\frac{(V, P) \xrightarrow{\checkmark} (V, P'), (V, Q) \xrightarrow{\checkmark} (V, Q')}{(V, P \parallel Q) \xrightarrow{\checkmark} (V, Stop)} \quad [\#_par2]$$

In addition, CSP_M allows the termination of a paralleled process to be independent of its associated process. Firing rules $[M_par1]$ below describes that if process P performs a tick action to P' , which means it has terminated, then the termination of the parallel process involves an invisible event τ and P becomes Ω denoting the termination of P ; operator \parallel_X is a general form of three kinds of parallel operators in CSP_M .

$$\frac{P \xrightarrow{\tau} P'}{P \parallel_X Q \xrightarrow{\tau} \Omega \parallel_X Q} \quad [M_par1] \qquad \frac{}{\Omega \parallel_X \Omega \xrightarrow{\tau} \Omega} \quad [M_par2]$$

The firing rule for Q is similar to $[M_par1]$. When both processes become Ω , the parallel process terminates under the firing rule $[M_par2]$. Notice that the verification results especially on non-terminating checking of parallel composition in CSP_M and $CSP\#$ are the same although the former needs two more steps.

Parallel processes involving synchronous channels in $CSP\#$ have been discussed early in Section 3.2.3 (by the firing rule $[\#_par1]$). Parallel processes involving asynchronous channels execute independently, described by 14 rules from $[\#_par3]$ to $[\#_par16]$. Firing rules $[\#_par3]$, $[\#_par4]$, $[\#_par5]$, $[\#_par6]$, $[\#_par7]$ and $[\#_par8]$ describe two asynchronous channel communications (input or output) run in parallel. ac_1 and ac_2 are asynchronous channel names, which may be the same or not.

$$\frac{ac_1 \text{ is not full in } V}{(V, ac_1!e \rightarrow P \parallel ac_2!e \rightarrow Q) \xrightarrow{ac_1!eva(V,e)} (app(V, ac_1!e), P \parallel ac_2!e \rightarrow Q)} \quad [\#_par3]$$

$$\frac{ac_2 \text{ is not full in } V}{(V, ac_1!e \rightarrow P \parallel ac_2!e \rightarrow Q) \xrightarrow{ac_2!eva(V,e)} (app(V, ac_2!e), ac_1!e \rightarrow P \parallel Q)} \quad [\#_par4]$$

$$\frac{ac_1 \text{ is not empty in } V \wedge (V \wedge x = top(ac_1)) \Rightarrow b}{(V, ac_1?[b]x \rightarrow P \parallel ac_2?[b]x \rightarrow Q) \xrightarrow{ac_1?top(ac_1)} (pop(V, ac_1?x), P[top(ac_1)/x] \parallel ac_2?[b]x \rightarrow Q)} \quad [\#_par5]$$

$$\frac{ac_2 \text{ is not empty in } V \wedge (V \wedge x = top(ac_2)) \Rightarrow b}{(V, ac_1?[b]x \rightarrow P \parallel ac_2?[b]x \rightarrow Q) \xrightarrow{ac_2?top(ac_2)} (pop(V, ac_2?x), ac_1?[b]x \rightarrow P \parallel Q[top(ac_2)/x])} \quad [\#_par6]$$

$$\frac{ac_1 \text{ is not full in } V}{(V, ac_1!e \rightarrow P \parallel ac_2?[b]x \rightarrow Q) \xrightarrow{ac_1!eva(V,e)} (app(V, ac_1!e), P \parallel ac_2?[b]x \rightarrow Q)} \quad [\#_par7]$$

$$\frac{ac_2 \text{ is not empty in } V \wedge (V \wedge x = top(ac_2)) \Rightarrow b}{(V, ac_1!e \rightarrow P \parallel ac_2?[b]x \rightarrow Q) \xrightarrow{ac_2?top(ac_2)} (pop(V, ac_2?x), ac_1!e \rightarrow P \parallel Q[top(ac_2)/x])} \quad [\#_par8]$$

Firing rules $[\#_par9]$, $[\#_par10]$, $[\#_par11]$ and $[\#_par12]$ describe asynchronous channel output/input run in parallel with synchronous channel output/input. Note that c denotes synchronous channel name and ac denotes asynchronous channel name.

$$\frac{ac \text{ is not full in } V}{(V, ac!e \rightarrow P \parallel c!e \rightarrow Q) \xrightarrow{ac!eva(V,e)} (app(V, ac!e), P \parallel c!e \rightarrow Q)} \quad [\#_par9]$$

$$\frac{ac \text{ is not empty in } V \wedge (V \wedge x = top(ac)) \Rightarrow b}{(V, ac?[b]x \rightarrow P \parallel c!e \rightarrow Q) \xrightarrow{ac?top(ac)} (pop(V, ac?x), P[top(ac)/x] \parallel c!e \rightarrow Q)} \quad [\#_par10]$$

$$\frac{ac \text{ is not full in } V}{(V, ac!e \rightarrow P \parallel c?[b]x \rightarrow Q) \xrightarrow{ac!eva(V,e)} (app(V, ac!e), P \parallel c?[b]x \rightarrow Q)} \quad [\#_par11]$$

$$\frac{ac \text{ is not empty in } V \wedge (V \wedge x = top(ac)) \Rightarrow b}{(V, ac?[b]x \rightarrow P \parallel c?[b]x \rightarrow Q) \xrightarrow{ac?top(ac)} (pop(V, ac?x), P[top(ac)/x] \parallel c?[b]x \rightarrow Q)} \quad [\#_par12]$$

Firing rules $[\#_par13]$, $[\#_par14]$, $[\#_par15]$ and $[\#_par16]$ describe asynchronous channel output/input runs in parallel with an event prefixing process or a data operation (denoted by process Q).

$$\frac{ac \text{ is not full in } V}{(V, (ac!e \rightarrow P) \parallel Q) \xrightarrow{ac!eva(V,e)} (app(V, ac!e), P \parallel Q)} \quad [\#_par13]$$

$$\frac{ac \text{ is not empty in } V \wedge (V \wedge x = top(ac)) \Rightarrow b}{(V, (ac?[b]x \rightarrow P) \parallel Q) \xrightarrow{ac?top(ac)} (pop(V, ac?x), P[top(ac)/x] \parallel Q)} \quad [\#_par14]$$

$$\frac{(V, Q) \xrightarrow{a} (V, Q'), a \notin \alpha P \cap \alpha Q}{(V, (ac!e \rightarrow P) \parallel Q) \xrightarrow{a} (V, (ac!e \rightarrow P) \parallel Q')} \quad [\#_par15]$$

$$\frac{(V, Q) \xrightarrow{a} (V, Q'), a \notin \alpha P \cap \alpha Q}{(V, (ac?[b]x \rightarrow P) \parallel Q) \xrightarrow{a} (V, (ac?[b]x \rightarrow P) \parallel Q')} \quad [\#_par16]$$

3.2.6 Interleaving

In CSP_M , interleaving process $P \parallel\parallel Q$ executes processes P and Q completely independently. The interleaving process in $CSP\#$ also executes P and Q independently, except the circumstances which involve communication through *shared variables* and synchronous channels. The firing rule on interleaving processes with synchronous channel communication

is as follows.

$$\frac{\begin{array}{l} (V, c!e \rightarrow P) \xrightarrow{c!eva(V,e)} (V, P), (V, c?[b]x \rightarrow Q) \xrightarrow{c?[b]x} (V, Q), \\ (V \wedge x = eva(V, e)) \Rightarrow b \end{array}}{(V, c!e \rightarrow P \parallel c?[b]x \rightarrow Q) \xrightarrow{c.eva(V,e)} (V, P \parallel Q[eva(V, e)/x])} \quad [\#_int]$$

3.2.7 General choice

CSP_M and $CSP\#$ have the same interpretation of *external choice* which is resolved by the occurrence of a *visible* event and *internal choice* which is resolved *nondeterministically*. Additionally, $CSP\#$ supports *general choice* $P \parallel Q$ which is resolved by any event, either invisible or visible. We present the firing rules below associated to the general choice in $CSP\#$ where $a \in \Sigma \cup \{\checkmark, \tau\}$.

$$\frac{(V, P) \xrightarrow{a} (V', P')}{(V, P \parallel Q) \xrightarrow{a} (V', P')} \quad [\#_gen1] \qquad \frac{(V, Q) \xrightarrow{a} (V', Q')}{(V, P \parallel Q) \xrightarrow{a} (V', Q')} \quad [\#_gen2]$$

3.2.8 Conditional choice

$CSP\#$ supports three types of conditional choice: 1) classic conditional choice *if* $b \{P\}$ *else* $\{Q\}$ executes P if b is evaluated to be true, and else performs Q (see rules $[\#_con1]$ and $[\#_con2]$), 2) atomic conditional choice *ifa* $b \{P\}$ *else* $\{Q\}$ behaves similarly to classic choice except that the condition checking is conducted *atomically* with the occurrence of the first event in P or Q (see rule $[\#_con3]$ and $[\#_con4]$), and 3) blocking conditional choice *ifb* $b \{P\}$ is similar to Boolean guards $[b]P$ in $CSP\#$ (which is equivalent to b&P in CSP_M) except that the checking of blocking condition and the execution of P are separated in *ifb* (see rule $[\#_con5]$). We remark that CSP_M supports only one type of conditional choice, which is equivalent to atomic conditional choice in $CSP\#$.

$$\frac{V \models b}{(V, \text{if } b \{P\} \text{ else } \{Q\}) \xrightarrow{\tau} (V, P)} \quad [\#_con1]$$

$$\begin{array}{c}
 \frac{V \not\models b}{(V, \text{if } b \{P\} \text{ else } \{Q\}) \xrightarrow{\tau} (V, Q)} \quad [\#_con2] \\
 \\
 \frac{V \models b, (V, P) \xrightarrow{a} (V', P'), a \in \Sigma \cup \{\checkmark, \tau\}}{(V, \text{if } a \{P\} \text{ else } \{Q\}) \xrightarrow{a} (V', P')} \quad [\#_cond3] \\
 \\
 \frac{V \not\models b, (V, Q) \xrightarrow{a} (V', Q'), a \in \Sigma \cup \{\checkmark, \tau\}}{(V, \text{if } a \{P\} \text{ else } \{Q\}) \xrightarrow{a} (V', Q')} \quad [\#_cond4] \\
 \\
 \frac{V \models b}{(V, \text{if } b \{P\}) \xrightarrow{\tau} (V, P)} \quad [\#_cond5]
 \end{array}$$

3.2.9 Renaming

CSP_M supports *renaming* which renames a visible event when an associated process is running, shown in the rule $[M_r3]$. In theory, event renaming $P[[R]]$ can be represented in $CSP\#$ by a process Q which is *almost* the same as P except the visible event from relation R being replaced. However, modelling the renaming process manually in $CSP\#$ may not be easy when the renaming relation is complicated, and it may lead to larger specifications in terms of line of code (LOC).

$$\begin{array}{c}
 \frac{P \xrightarrow{\tau} P'}{P[[R]] \xrightarrow{\tau} P'[[R]]} \quad [M_r1] \qquad \frac{P \xrightarrow{\checkmark} P'}{P[[R]] \xrightarrow{\checkmark} \Omega} \quad [M_r2] \\
 \\
 \frac{P \xrightarrow{a} P', a \ R \ b, a, b \in \Sigma}{P[[R]] \xrightarrow{b} P'[[R]]} \quad [M_r3]
 \end{array}$$

3.2.10 Untimed timeout

Process *untimed timeout* $P[> Q$ in CSP_M executes P for an unspecified amount of time followed by performing Q . The firing rules shown below specify that the untimed timeout process initially allows P to decide the choice with any visible action ($[M_unt2]$), while at any moment it can timeout, and becomes to Q ($[M_unt3]$), or if P performs an invisible event τ to P' , then $P[> Q$ will perform a τ transition to $P'[> Q$ as timeout is not resolved by τ action ($[M_unt1]$).

$$\begin{array}{c}
 \frac{P \xrightarrow{\tau} P'}{P[> Q \xrightarrow{\tau} P'[> Q} \quad [M_unt1] \qquad \frac{P \xrightarrow{a} P', a \in \Sigma}{P[> Q \xrightarrow{a} P'} \quad [M_unt2] \\
 \\
 \frac{}{P[> Q \xrightarrow{\tau} Q} \quad [M_unt3]
 \end{array}$$

3.2.11 Discussion

We have identified differences between CSP_M and $CSP\#$ in terms of their operational semantics, and also discussed some possible translations between these two languages, especially their channel communications. Through the analysis, we can draw some general guidelines of their modelling features: CSP_M 's adoption of functional paradigm and support of more primitives such as *CHAOS* and *renaming* provide an approach to specify concurrent systems, starting with an abstract model first, which can then be refined to more concrete one. $CSP\#$ supports more primitives for modelling different forms of communication, and it is feasible to specify concrete system behaviours which require hand shaking, message passing and shared resources. In term of expressiveness, it can be shown that CSP_M and $CSP\#$ are equivalent as both CSP_M and $CSP\#$ process can be transformed into a normal form, which involves event-prefixing, internal choice and recursion only [62].

3.3 Verification Tool Support

CSP_M is supported by FDR which is designed primarily for refinement checking in terms of trace, failures, divergences, refusals and revivals. ProB was initially designed as an animator and model checker for B method [2], and recently it supports CSP_M with improvements on static type checking and associative tuples [38]; ProB integrates type checking, animation and model checking together. $\text{CSP}\#$ is supported by PAT which is an extensible framework for system modelling, simulation and verification. PAT implements a number of model checking techniques catering for different properties such as LTL properties and refinement checking [82, 40, 41]. In the following, Section 3.3.1 illustrates the verification capabilities of FDR, ProB (for CSP_M) and PAT (for $\text{CSP}\#$), including properties supported and their model checking techniques; Section 3.3.2 investigates the efficiency of the three tools.

3.3.1 Verification

FDR, ProB and PAT support the analysis of many common properties such as deadlock, livelock, determinism, and refinement checking which includes trace, failure, and failures/-divergences refinement. In addition, FDR supports two additional refinement models: the refusal testing model and the revivals model [45]. In the refusal testing model [43], a process is represented by a sequence of alternating refusal sets and events, possibly terminating in deadlock where the refusal set are a set of all events. In unstable states where no proper refusal can be observed, the refusal set is represented by a special null refusal value. In the revivals model [64], a process is represented by a tuple with three elements, the process's finite traces, the finite traces on which it can deadlock and a set of revivals. A revival is of the form (s, X, a) denoting that the process might accept event a after completing the trace s , where X is a stable refusal set observed before event a happens. PAT supports additional properties like *reachability analysis*, i.e., if a system can reach a bad state (e.g.,

array overflow).

Model checking LTL properties is common in practice. Although it is not directly supported in FDR, the relationship between refinement checking and LTL model checking has been studied (e.g., [63, 51]). Particularly, Leuschel et al. [39] applied an emptiness test in a refinement between an unexpected specification and a process; the process is a synchronisation of the implementation and a CSP process for an LTL formula. This approach has to deal with the high complexity of synchronisation in FDR, and the process to construct CSP processes from LTL formulas is arduous. Lowe [43] used a refusal testing model to conduct the refusal refinement between a CSP process which denotes an LTL formula and its implementation; those supported LTL formulas exclude operators eventually (\diamond), until (\mathcal{U}), and negation. In contrast, ProB and PAT support various LTL formulas and analysis directly. Moreover, these formulas can constrain both states and events, and be analysed under five types of fairness assumptions [85] in PAT.

FDR, ProB, and PAT all provide basic model checking techniques such as breadth first search and (bounded) depth first search. In addition, PAT implements the anti-chain approach in which the complete subset construction and computing the complete state space of the product are avoided for checking refinement [90]. Further, PAT applies Loop/SCC searching algorithm for LTL verification under fairness assumptions. To cope with the problem of state space explosion during verification, FDR and PAT develop their own reduction techniques. To be specific, FDR proposes a hierarchical compression approach consisting of six methods to process an LTS representing a CSP_M model [45, 62, 66]: enumerations, strongly node-labelled bisimulation, τ -loop elimination, diamond elimination, normalization, and factoring by semantic equivalence. On the other hand, PAT deploys three techniques. First, using the atomic sequence construct (denoted by $\text{atomic}\{P\}$), where a sequence of statements in a process executes as one *super-step* without any inference, to realize simple partial order reduction (POR). Second, applying POR dedicated to refinement checking to not only

τ transitions but also visible events (in some case which is not supported in FDR [85]). Last but not least, providing process counter abstraction for parameterized systems under fairness against LTL formulas [87]. We remark that the implementation of FDR’s hierarchical compression methods for CSP\# in PAT is nontrivial due to shared variables supported in CSP\# . For instance, a τ event in CSP\# may update shared variables and therefore the event cannot not be pruned for compression.

3.3.2 Experiment

In this section, we evaluate the efficiency of FDR, ProB and PAT by verifying eight benchmark systems: readers/writers (R/W), dining philosophers problem (DP), Milner’s cyclic scheduler (MCS) [49], peg solitaire puzzle, chess knight tour puzzle, the tower of Hanoi puzzle, concurrent stack [41] and Peterson’s algorithm [59]. The experiments with FDR and ProB are performed on an Intel[®] CPU E6550 (2.33 GHz) PC with 4GB memory running on 32-bit Linux. PAT is experimented with the same PC but on a 32-bit Windows.

We conduct four sets of experiments. The CSP\# models for these benchmark systems are in Appendix A. The first set investigates the performance of refinement checking, by verifying the same model and assertion with different reduction techniques. The results are shown in Table 3.2, where N is the number of processes. Column *State* shows the number of visited states, and column *Time(s)* records running time of the verification in seconds. Value “-” in a cell denotes that the experiment is aborted due to either memory overflow or execution time exceeding two hours. For readers/writers (R/W) models, although FDR applies some dedicated compression techniques, PAT has better performance. For dining philosopher (DP) models, FDR performs extremely well because of the strategy discussed in [67]. However, other experiments show that this strategy may not be as efficient for other models. For Milner’s cyclic scheduler (MCS), PAT is comparable to FDR in terms of the number of states per second. FDR processes the LTS by applying its compression methods,

whereas PAT applies a simple reduction method, i.e., using the keyword *atomic* to give higher priority to local events which are not synchronised, not updating any variable and not mentioned in the property.

Model	N	Property	FDR		ProB		PAT	
			State	Time(s)	State	Time(s)	State	Time(s)
R/W	6	P [T= S	8	0.024	61365	125.94	9	0.04
R/W	200	P [T= S	202	1.434	-	-	203	0.11
R/W	500	P [T= S	502	19.651	-	-	503	0.057
R/W	1000	P [T= S	1002	156.162	-	-	1003	0.108
DP	6	P [F= S	1	0.06	14510	82.42	1762	0.174
DP	8	P [F= S	1	0.071	-	-	22362	2.995
DP	12	P [F= S	1	0.104	-	-	-	-
MCS	20	P [FD= S	40	0.043	-	-	60	0.114
MCS	50	P [FD= S	100	0.086	-	-	150	0.143
MCS	100	P [FD= S	200	0.246	-	-	300	0.53

Table 3.2: Experiment results on refinement checking

The second set compares the performance of three model checkers on solving puzzles, inspired by the work in [56]. The CSP_M and $CSP_\#$ models for these puzzles make the best use of their modelling power: $CSP_\#$ specifies the puzzles using shared variables, which are solved by PAT through reachability analysis, whereas CSP_M models the puzzles using multi-part event synchronisation, which are solved by FDR and ProB through trace refinement. In addition, FDR simulates a bounded DFS algorithm by searching the divergence of a new system, in order to find a smaller counterexample. This new system, like a watchdog, can only perform up to N events of the target implementation process, and then performs an infinite number of events [56]; this approach can be used provided that the target process is loop-free. Table 3.3 shows the performance results, where column *FDR-Div* records the results of states and time using this algorithm; value *N.A.* means there is no model with divergence checking to solve the puzzle. From Table 3.3, we can observe that the divergence checking approach can be used in the solitaire and chess knight tour models. However, this approach cannot always significantly improve performance, because it depends on the

searching order. Moreover, it is costly to check if a system is loop-free or not, which is the premise for applying this approach. PAT solves the two puzzles in a reasonable time, and it is faster in the knight example than FDR and FDR-Div. For the Hanoi puzzle, FDR has a better performance because the compression techniques it uses can effectively reduce the state space.

Model	N	FDR		FDR-Div		ProB		PAT	
		State	Time(s)	State	Time(s)	State	Time(s)	State	Time(s)
Solitaire	26	4048216	46.303	1	0.169	-	-	11950	5.356
Solitaire	29	28249254	387.737	1	0.217	-	-	104395	54.681
Solitaire	32	-	-	1	5.318	-	-	10955	5.301
Solitaire	35	-	-	1	377.297	-	-	443230	279.454
Knight	5	508450	3.522	1	0.037	-	-	4256	0.29
Knight	6	-	-	1	15.399	-	-	129269	9.143
Knight	7	-	-	1	94.713	-	-	77238	6.754
Hanoi	6	729	0.052	N.A.	N.A.	1667	57.84	5775	0.416
Hanoi	7	2187	0.086	N.A.	N.A.	4969	196.5	92680	6.837
Hanoi	8	6561	0.181	N.A.	N.A.	14853	660.59	150918	11.524

Table 3.3: Experiment results on solving puzzles

The third set explores the performance of FDR and PAT on verifying two models which involve shared variables. The first example is a concurrent stack which allows multiple readers to access the shared variable at the same time, but only one writer to update the value; readers cannot access the shared variable when it is written. The modelling of shared variables in CSP_M follows the approach discussed in Section 3.2.4. Results of this example in Table 3.4 show that PAT performs better than FDR for checking trace refinement ($P \sqsubseteq T = S$), and this is because PAT uses DFS with anti-chain algorithm in the trace refinement. This algorithm is effective when the specification is non-deterministic. Here, N is the number of processes and *ConcurrentStack* * 2 in the *Model* column means that the stack size is 2. The second example is the Peterson algorithm. We obtain the CSP_M model from the shared variable analyser (SVA) [66]. To be fair, the $\text{CSP}\#$ model is specified at the same level of granularity as the CSP_M model. The results show that PAT performs better. This is because local events associated as atomic statements in $\text{CSP}\#$ reduce the states

significantly, whereas CSP_M model defines additional events to represent reading/writing operations of shared variables. Although these additional events can be hidden as internal events to apply existing compression techniques in FDR, the effect is minor because the type range of reading/writing channels and operations over different variables can easily lead to state space explosion.

Model	N	Property	FDR		PAT	
			State	Time(s)	State	Time(s)
Concurrent Stack*2	3	P [T= S	453456	3.833	10860	1.023
Concurrent Stack*2	4	P [T= S	-	-	189920	75.915
Concurrent Stack*2	5	P [T= S	-	-	693828	293.382
Peterson	3	mutual exclusion	1011	1.192	3257	0.105
Peterson	4	mutual exclusion	105493	20.067	104686	3.776
Peterson	5	mutual exclusion	14810779	387.645	5722863	294.005

Table 3.4: Experiment results on shared variables

The fourth set explores the performance on verifying LTL properties. We adopt the approach proposed by Lowe [43] to construct a CSP_M process for the LTL formula and use FDR to perform the refusal refinement checking. As this approach cannot deal with operator *eventually* (\diamond), we ignore the checking of property $\Box \diamond eat.0$ in FDR. Table 3.5 indicates that PAT performs better than FDR and ProB. Notice that property $\Box \diamond eat.0$ can be verified to be true using PAT under the strong or global fairness assumption.

Model	N	Property	Result	FDR		ProB		PAT	
				State	Time(s)	State	Time(s)	State	Time(s)
RW	6	$\Box !error$	true	8	0.023	122722	104.8	15	0.059
RW	200	$\Box !error$	true	202	1.455	-	-	403	0.086
RW	500	$\Box !error$	true	502	19.901	-	-	1003	0.071
RW	1000	$\Box !error$	true	1002	154.33	-	-	2003	0.148
DP	6	$\Box \diamond eat.0$	false	N.A.	N.A.	2420	1.11	166	0.019
DP	8	$\Box \diamond eat.0$	false	N.A.	N.A.	13312	1.75	256	0.024
DP	12	$\Box \diamond eat.0$	false	N.A.	N.A.	-	-	460	0.049

Table 3.5: Experiment results on LTL checking

3.3.3 Discussion

We have explored the supporting tools of CSP_M and $\text{CSP}_\#$, namely, FDR, ProB and PAT, by comparing their model checking techniques and analysing their verification capabilities through eight benchmark systems. Our exploration leads to the following four general and practical rules for choosing these tools. First, FDR can be the best candidate when powerful built-in compression techniques are applicable in refinement checking. Second, PAT is a better choice to verify properties of models which involve shared variables. Third, to verify LTL properties, we can use ProB for CSP_M models or FDR for some model where LTL formula can be verified by refusal checking, and PAT for $\text{CSP}_\#$ model. Lastly, PAT may be a better option to handle models where atomic reductions are applicable (e.g., readers/writers and Peterson algorithm).

3.4 Summary

In this chapter, we presented a comprehensive comparison of CSP_M and $\text{CSP}_\#$, and their supporting tools FDR, ProB and PAT. We explored their modelling features from the view of their syntax and operational semantics. We also investigated the reasoning power of CSP_M and $\text{CSP}_\#$ in terms of the capability and efficiency of their supporting tools. We derive the following guidelines from our comparison for selecting appropriate modelling language and reasoning tools for particular concurrent systems. 1) CSP_M may be more suitable to model systems with abstract behaviour, and systems which involve multi-part event synchronisation. On the other hand, $\text{CSP}_\#$ could be a better candidate to handle systems which implement hand shaking or message passing communication mechanisms, and systems which need shared variables. 2) To perform the refinement checking, the decision relies on the reduction techniques which are more applicable (compression methods in FDR, atomic reduction in PAT) to the models. To verify LTL properties, we can use ProB for CSP_M

models or FDR for some model (discussed in Section 3.3), and PAT for $\text{CSP}_\#$ models. Lastly, PAT may be a better option to verify systems with shared variables.

As for related work, Carvalho et al. [13] have made an initial step to explore the differences between CSP_M and $\text{CSP}_\#$. They compared the two languages from the data and behavioural aspects. Our work here substantially extends their effort by an in-depth and a wider range of comparisons; for instance, we investigate their intrinsic differences from the operational semantics aspect. Roscoe [1] has briefly described tools which can animate, analyse, and verify CSP models; these tools include FDR, ProB, PAT, ARC [57] and so on. He introduces these tools with strengths and limits from a high level. Our work can be considered as a concrete guideline for these tools, in particular, FDR, ProB for CSP_M , and PAT for $\text{CSP}_\#$, with intensive experiments.

Chapter 4

A UTP Semantics for CSP#

Formal semantics of a specification language represents the mathematically precise and rigorous meaning of expressions in that language. This semantics thus provides a foundation to analyse system behaviours or system design described in such a language. Formal semantics is usually classified into three groups, operational, denotational and axiomatic. As shown in Chapter 3, the small step operational semantics of CSP# has been formally defined, which is used in PAT as the execution firing rules [83]. The operational semantics in general has some limitations in the following aspects. First, operational semantics is hard to compare programs in different specification languages. Second, operational semantics cannot specify conditions under which two different programs written in the same language are equal or one refines the other. Third, operational semantics cannot specify how transition rules are selected. For example, is the choice made in advance or at run time? In addition, the defined operational semantics for CSP# is not compositional, and thus lacks the support of compositional verification of process behaviours. Therefore, there is a need for a compositional denotational semantics to explain the notations of the CSP# languages.

The challenge of defining a denotational semantics for CSP# is to develop an appropriate

semantic model which can cover not only communications but also shared variable paradigm. The Unifying Theories of Programming (UTP) [31] is a unified framework for defining denotational semantics for programs across different programming paradigms, suitable for defining the denotational semantics for CSP#.

In this chapter, we present an observation-oriented denotational semantics for the CSP# language based on the UTP framework in an *open* environment, where process behaviours can be interfered with by the environment. The proposed semantics not only provides a rigorous meaning of the language, but also deduces algebraic laws describing the properties of CSP# processes. To deal with shared variables, we lift traditional event-based traces into *hybrid* traces (capturing both states and events) for recording process behaviours. To handle different types of synchronisation in CSP# (i.e., event-based and synchronised handshake), we construct a comprehensive set of rules on merging traces from processes which run in parallel/interleaving. These rules capture all possible concurrency behaviours between event/channel-based communications and global shared variables.

The contributions of our work are summarized as follows.

- The proposed semantic model deals with not only communicating processes, but also shared variables. It can model both event-based synchronisation and synchronised handshake over channels. Moreover, our model can be adapted/enhanced to define the denotational semantics for other languages which possess similar concurrency mechanisms.
- The defined denotational semantics overcomes the limitations of operational semantics mentioned above. For example, we define the refinement and equivalent relationship between programs written in CSP#.
- A *closed* semantics can be derived from our open denotational semantics by focusing on special types of hybrid traces. The closed semantics can be linked with the CSP#

operational semantics in [83].

The remainder of this chapter is organized as follows. Section 4.1 constructs the observation-oriented denotational semantics in an open environment based on the UTP framework; healthiness conditions are also defined to characterise the semantic domain. Section 4.2 discusses the algebraic laws for CSP#. Section 4.3 presents a closed semantics derived from the open semantics. Section 4.4 concludes this chapter with the discussion of related work.

4.1 Denotational Semantics of CSP#

In this section, we first define the semantic model including the observational variables and healthiness conditions. We then define the denotational semantics of arithmetic and Boolean expressions as well as the denotational semantics of sequential programs. Based on the semantic model and the semantics of expressions and programs, we define the denotational semantics of CSP# processes.

4.1.1 Semantic Model

To address the challenge of designing an appropriate CSP# semantic model to cover both communication and shared variables paradigms, we blend communication events with states containing shared variables. Namely, we introduce *hybrid* traces to record the interactions of processes with the global environment; each trace is a sequence of pairs, and each pair is either a pair of shared variable states or comprised of a pre-state and a communication event.

4.1.1.1 Observational Variables

The following variables are introduced in the alphabet of observations of CSP# process behaviour. Some of them, i.e., ok , ok' , $wait$, $wait'$, ref , and ref' , are similar to those in the UTP theory for CSP [31]. The key difference is that the event-based traces in CSP are changed to hybrid traces capturing both states and events.

- ok , ok' : Boolean describe the stability of a process.
 $ok = true$ records that the process has started in a stable state, whereas $ok = false$ records that the process has not started as its predecessor has diverged.
 $ok' = true$ records that the process has reached a stable state, whereas $ok' = false$ records that the process has diverged.
- $wait$, $wait'$: Boolean distinguish the intermediate observations of waiting states from the observations of final states.
 $wait = true$ records that the execution of the previous process has not finished, and the current process starts in an intermediate state, while $wait = false$ records that the execution of the previous process has finished and the current process may start.
 $wait' = true$ records that the next observation of the process is in an intermediate state, while $wait' = false$ records that the next observation is in a terminated state.
- ref , ref' : $\mathbb{P}Event$ denote a set of actions and channel inputs/outputs that can be refused before or after the observation. The set $Event$ denotes all possible actions and channel input/output directions (e.g., $ch?$, $ch!$). An input direction $ch?$ denotes any input through channel ch , and a channel output direction $ch!$ denotes any output through channel ch .
- tr , tr' : $seq((S \times S^\perp) \cup (S \times E))$ record a sequence of observations (state pairs or communication events) on the interaction of processes with the global environment.

- S is the set of all possible mappings (states), and a state $s : \text{VAR} \rightarrow T$ is a function which maps global shared variables VAR into values of T . Notice that we use T to denote the types of variables and channel messages.
- E is the set of all possible events, including actions, channel inputs/outputs and synchronous channel communications. Note that non-communicating events are excluded from the set.
- $S \times S^\perp$ is the set of state pairs, and each pair consists of a pre-state recording the initial variable values before the observation and a post-state recording the final values after the observation. $S^\perp \triangleq S \cup \{\perp\}$ represents all states, where the improper state \perp indicates non-termination. Remark that the state pair is used to record the observation for the sequential program.
- $S \times E$ denotes a set of occurring events under the pre-states. The reason of recording the pre-state is that the value of the expression which may contain shared variables in a channel output shall be evaluated under this state.

4.1.1.2 Healthiness Conditions

Healthiness conditions are defined as equations in terms of an idempotent function ϕ on predicates. Every healthy program represented by predicate P must be a fixed point under the healthiness condition of its respective UTP theory, i.e., $P = \phi(P)$.

In CSP#, a process can never change the past history of the observations; instead, it can only extend the record, captured by function **R1**. We use predicate P to represent the semantics of the CSP# process below.

$$\mathbf{R1}: \quad \mathbf{R1}(P) = P \wedge tr \leq tr'$$

The execution of a process is independent of the history before its activation, captured by function **R2**.

$$\mathbf{R2}: \quad \mathbf{R2}(P(tr, tr')) = \sqcap_s P(s, s \frown (tr' - tr))$$

As mentioned earlier, variable *wait* distinguishes a waiting state from the final state. A process cannot start if its previous process has not finished, or otherwise, the values of all observational variables are unchanged, characterised by function **R3**.

$$\mathbf{R3}: \quad \mathbf{R3}(P) = II \triangleleft wait \triangleright P$$

where $P \triangleleft b \triangleright Q \hat{=} b \wedge P \vee \neg b \wedge Q$ and $II \hat{=} (\neg ok \wedge tr \leq tr') \vee (ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$. Here *II* states that if a process is in a divergent state, then only the trace can be extended, or otherwise, it is in a stable state, and the values of all observational variables remain unchanged.

When a process is in a divergent state, it can only extend the trace. This feature is captured by function **CSP1**.

$$\mathbf{CSP1}: \quad \mathbf{CSP1}(P) = (\neg ok \wedge tr \leq tr') \vee P$$

Every process is monotonic in the observational variable *ok'*. This monotonicity property is modelled by function **CSP2** which states that if an observation of a process is valid when *ok'* is false, then the observation should also be valid when *ok'* is true.

$$\mathbf{CSP2}: \quad \mathbf{CSP2}(P) = P; (ok \Rightarrow ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$$

We below use **H** to denote all healthiness conditions satisfied by the CSP# process.

$$\mathbf{H} = \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3} \circ \mathbf{CSP1} \circ \mathbf{CSP2}$$

From the above definition, we can see that although CSP# satisfies the same healthiness conditions of CSP, observational variables *tr*, *tr'* in our semantic model record additional information for shared variable states. We adopt the same names for the idempotent functions used in CSP for consistency. In addition, function **H** is idempotent and monotonic [14, 31].

4.1.2 Semantics of Expressions and Programs

In this section, we first present the syntax of a subset of arithmetic expressions, Boolean expressions and sequential programs, and next define their denotational semantics which will be used in Section 4.1.3.

$$\begin{array}{ll}
 \text{prog} ::= x = \text{exp} & \text{-- assignment} \\
 \quad | \text{prog}_1; \text{prog}_2 & \text{-- composition} \\
 \quad | \text{if } b \text{ then } \text{prog}_1 \text{ else } \text{prog}_2 & \text{-- conditional} \\
 \quad | \text{while } b \text{ do } \text{prog} & \text{-- iteration} \\
 \text{exp} ::= v \mid x \mid \text{exp}_1 + \text{exp}_2 \mid \text{exp}_1 - \text{exp}_2 \mid \text{exp}_1 * \text{exp}_2 \mid \text{exp}_1 / \text{exp}_2 \\
 b ::= \text{true} \mid \text{false} \mid \text{exp}_1 \text{ op } \text{exp}_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
 \text{where op} \in \{=, \neq, <, \leq, >, \geq\}
 \end{array}$$

In the above syntax representation, x is a global shared variable, v is a value, exp , exp_1 and exp_2 are arithmetic expressions, b , b_1 and b_2 are Boolean expressions, and prog , prog_1 and prog_2 are sequential programs updating global shared variables.

Definition 1 (Arithmetic Expression). *Let Aexp be the type of arithmetic expressions, the evaluation of the expression is defined as a function $\mathcal{A} : \text{Aexp} \rightarrow (\text{S} \rightarrow \text{T})$.*

$$\begin{aligned}
 \mathcal{A}[\![n]\!](s) &= n \\
 \mathcal{A}[\![x]\!](s) &= s(x) \\
 \mathcal{A}[\![\text{exp}_1 + \text{exp}_2]\!](s) &= \mathcal{A}[\![\text{exp}_1]\!](s) + \mathcal{A}[\![\text{exp}_2]\!](s) \\
 \mathcal{A}[\![\text{exp}_1 - \text{exp}_2]\!](s) &= \mathcal{A}[\![\text{exp}_1]\!](s) - \mathcal{A}[\![\text{exp}_2]\!](s) \\
 \mathcal{A}[\![\text{exp}_1 * \text{exp}_2]\!](s) &= \mathcal{A}[\![\text{exp}_1]\!](s) * \mathcal{A}[\![\text{exp}_2]\!](s) \\
 \mathcal{A}[\![\text{exp}_1 / \text{exp}_2]\!](s) &= \mathcal{A}[\![\text{exp}_1]\!](s) / \mathcal{A}[\![\text{exp}_2]\!](s)^1
 \end{aligned}$$

Definition 2 (Boolean Expression). *Let Bexp be the type of Boolean expressions, given a valuation, function \mathcal{B} returns whether a boolean expression is valid, defined as $\mathcal{B} : \text{Bexp} \rightarrow (\text{S} \rightarrow \text{Boolean})$.*

¹We assume the expression is well-defined (i.e., $\mathcal{A}[\![\text{exp}_2]\!](s) \neq 0$).

$$\begin{aligned}
\mathcal{B}[\text{true}](s) &= \text{true} \\
\mathcal{B}[\text{false}](s) &= \text{false} \\
\mathcal{B}[\text{exp}_1 \text{ op exp}_2](s) &= \begin{cases} \text{true} & \mathcal{A}[\text{exp}_1](s) \text{ op } \mathcal{A}[\text{exp}_2](s) \\ \text{false} & \text{otherwise} \end{cases} \\
\mathcal{B}[\neg b](s) &= \neg(\mathcal{B}[b](s)) \\
\mathcal{B}[b_1 \wedge b_2](s) &= \mathcal{B}[b_1](s) \wedge \mathcal{B}[b_2](s) \\
\mathcal{B}[b_1 \vee b_2](s) &= \mathcal{B}[b_1](s) \vee \mathcal{B}[b_2](s)
\end{aligned}$$

Definition 3 (Sequential Program). Let Prog be the type of sequential programs, function \mathcal{C} returns the updated valuations after executing the program, defined as $\mathcal{C} : \text{Prog} \rightarrow (\mathcal{S} \rightarrow \mathcal{S}^\perp)$.

$$\begin{aligned}
\mathcal{C}[x := \text{exp}] &= \{(s, s[n/x]) \mid s \in \mathcal{S} \wedge n = \mathcal{A}[\text{exp}](s)\} \\
\mathcal{C}[\text{prog}_1; \text{prog}_2] &= \{(s, s') \mid \exists s_0 \in \mathcal{S} \bullet (s, s_0) \in \mathcal{C}[\text{prog}_1] \\
&\quad \wedge (s_0, s') \in \mathcal{C}[\text{prog}_2]\} \cup \\
&\quad \{(s, \perp) \mid (s, \perp) \in \mathcal{C}[\text{prog}_1]\} \\
\mathcal{C}[\text{if } b \text{ then } \text{prog}_1 \text{ else } \text{prog}_2] &= \{(s, s') \mid \mathcal{B}[b](s) = \text{true} \wedge (s, s') \in \mathcal{C}[\text{prog}_1]\} \cup \\
&\quad \{(s, s') \mid \mathcal{B}[b](s) = \text{false} \wedge (s, s') \in \mathcal{C}[\text{prog}_2]\} \\
\mathcal{C}[\text{while } b \text{ do } \text{prog}] &= \{(s, s') \mid (s, s') \in \mathcal{C}[\mu X \bullet F(X)]\}
\end{aligned}$$

In the above definition, $F(X) \triangleq \text{if } b \text{ then } \text{prog}; X \text{ else } \text{skip}$, $\mathcal{C}[\text{skip}] = \{(s, s) \mid s \in \mathcal{S}\}$, $\mathcal{C}[\text{true}] = \{(s, s') \mid s \in \mathcal{S}, s' \in \mathcal{S}^\perp\}$, and $\mu X \bullet F(X) \triangleq \bigcap_n F^n(\text{true})$.

4.1.3 Semantics of Processes

In this section, we construct an observation-oriented semantics for all CSP# process operators based on our proposed UTP semantic model for CSP#. We define the semantics in an open environment to achieve the compositionality property; namely, a process may be interfered with by the environment. In Section 4.1.1.1, we have defined a hybrid trace to record the potential events and state transitions in which a process P may engage; for example, the trace $tr' = \langle (s_1, s'_1) \rangle \cap \langle (s_2, a_2) \rangle$ describes the transitions of process P . In an open environment, tr' may contain an (implicit) transition (s'_1, s_2) as the result of interference by the environment where states s'_1 and s_2 can be different.

In the following, we illustrate our semantic definitions for the CSP# process operators, and present the refinement definition. Note that asynchronous channels and general choice are not discussed in this section; the former can be simulated easily with global variables, and the latter can be simulated by other CSP# operators, shown in Chapter 3. A denotational semantics for the interrupt operator in the Unifying Theory has been defined in [48].

4.1.3.1 Primitives

Deadlock process *Stop* never engages in any event or updates shared variables, and it is always waiting.

$$Stop \hat{=} \mathbf{H}(ok' \wedge tr' = tr \wedge wait')$$

The semantics shows that the trace is unchanged and process is in a waiting state (represented by *wait'* being true). In addition, *Stop* refuses all events, so the final value of the refusal set, *ref'*, is left unconstrained.

Process *Skip* terminates immediately without any event or state change occurring.

$$Skip \hat{=} \mathbf{H}(\exists ref \bullet II)$$

Reactive identity *II* constrains that if a process terminates, then there is no change on the trace. The initial refusal of *Skip* is irrelevant to its behaviour, defined by the existential quantifier. After termination, the refusal set *ref'* is arbitrary.

4.1.3.2 Event Prefixing

Process $a \rightarrow P$ engages in event *a* first and afterwards behaves as process *P*. Event *a* defined here is an action which occurs instantaneously, and may require simultaneous participation by more than one processes.

$$a \rightarrow P \triangleq \mathbf{H} \left(ok' \wedge \begin{pmatrix} a \notin ref' \wedge tr' = tr \\ \triangleleft wait' \triangleright \\ \exists s \in \mathbf{S} \bullet tr' = tr \cap \langle (s, a) \rangle \end{pmatrix} \right); P$$

The above semantics shows two possible behaviours: when a process is waiting to engage in action a , it cannot refuse this action during the waiting period (represented by predicate $a \notin ref'$), and its trace is unchanged; or a process performs action a and terminates with its trace extended with this observation (by predicate $tr' = tr \cap \langle (s, a) \rangle$). Since the environment may interfere with the process behaviour and make a transition on the shared variable states, we use state s from the variable state set \mathbf{S} to denote the initial state before the observation. Note that the semantics of sequential composition “;” is defined in Section 4.1.3.5.

4.1.3.3 Synchronous Channel Output/Input

In CSP#, messages can be sent/received synchronously through channels. The synchronisation is pairwise, involving two processes. Specifically, a synchronous channel communication $ch.exp$ can take place only if an output $ch!exp$ is enabled and a corresponding input $ch?m$ is also ready.

$$ch!exp \rightarrow P \triangleq \mathbf{H} \left(ok' \wedge \begin{pmatrix} ch? \notin ref' \wedge tr' = tr \\ \triangleleft wait' \triangleright \\ \exists s \in \mathbf{S} \bullet tr' = tr \cap \langle (s, ch!\mathcal{A}[\![exp]\!](s)) \rangle \end{pmatrix} \right); P$$

The above semantics of synchronous channel output depicts two possible behaviours: when a process is waiting to communicate on channel ch , it cannot refuse any channel input over ch provided by the environment to perform a channel communication (represented by predicate $ch? \notin ref'$), and its trace is unchanged; or a process performs the output through ch and terminates without divergence. The observation of the trace is recorded as a tuple

$(s, ch! \mathcal{A}[\![exp]\!](s))$, where the value of the output message is evaluated under the pre-state s . Here function \mathcal{A} defines the semantics of arithmetic expressions, and its definition is in Definition 1. After the output occurs, the process behaves as P .

$$ch?m \rightarrow P(m) \hat{=} \exists v \in \mathsf{T} \bullet \mathbf{H} \left(\left(ok' \wedge \begin{pmatrix} ch! \notin ref' \wedge tr' = tr \\ \triangleleft wait' \triangleright \\ \exists s \in \mathsf{S} \bullet tr' = tr \frown \langle (s, ch?v) \rangle \end{pmatrix} \right) ; P(v) \right)$$

As shown above, the semantics of synchronous channel input is similar to channel output except that when a process is waiting, it cannot refuse any channel output provided by the environment, and after the process receives a message v from channel ch , its trace is appended with a tuple $(s, ch?v)$. In addition, parameter m cannot be modified in process P ; namely, it becomes constant-like and its value is replaced by value v .

4.1.3.4 Data Operation Prefixing

In CSP#, sequential programs are executed atomically together with the occurrence of an event, called data operation. The updates on shared variables are observed after the execution of all programs as illustrated below.

$$e\{prog\} \rightarrow Skip \hat{=} \mathbf{H} \left(\left(ok' \wedge \exists s \in \mathsf{S} \bullet \begin{pmatrix} wait' \wedge tr' = tr \frown \langle (s, \perp) \rangle \\ \triangleleft (s, \perp) \in \mathcal{C}[\![prog]\!] \triangleright \\ \neg wait' \wedge \exists s' \in \mathsf{S} \bullet (tr' = tr \frown \langle (s, s') \rangle \\ \wedge (s, s') \in \mathcal{C}[\![prog]\!]) \end{pmatrix} \right) \right)$$

If the evaluation of the program does not terminate (represented by predicate $(s, \perp) \in \mathcal{C}[\![prog]\!]$), then the process is in a waiting state, and its trace is extended with the record of non-termination. On the other hand, if the evaluation succeeds and terminates, then the process terminates and the state transition is recorded in the trace. In our definition,

the non-communicating event is not recorded in the trace since such an event would not synchronise with other events; instead, its effect can be described by the updates on shared variable states. Thus the non-communicating event is used as a label to indicate the updates on shared variables. Note that post-state s' after the observation is associated with the pre-state s under the semantics of sequential programs $((s, s') \in \mathcal{C}[\![prog]\!])$. Function \mathcal{C} defines the semantics of programs by structured induction [94], its definition is in Section 4.1.2.

The data operation prefixing process $e\{prog\} \rightarrow P$ is thus defined as sequential composition of data operation and P .

$$e\{prog\} \rightarrow P \triangleq (e\{prog\} \rightarrow Skip); P$$

4.1.3.5 Sequential Composition

In process $P; Q$, P takes control first and Q starts only when P has finished.

$$P; Q \triangleq \exists obs_0 \bullet (P[obs_0/obs'] \wedge Q[obs_0/obs])$$

The semantics of sequential composition shows that if process P diverges, then so does the process $P; Q$; if process P is in a waiting state, then the following process Q cannot start; if P terminates, then process Q starts immediately and the final observation of process P is the initial observation of process Q . In the above definition, the term obs represents the set of observational variables ok , $wait$, tr , and ref , as is the case of obs_0 and obs' .

4.1.3.6 Choice

Internal choice denotes that process $P \sqcap Q$ behaves like either P or Q . The selection is made internally and non-deterministically, not affected by the environment.

$$P \sqcap Q \triangleq P \vee Q$$

External choice denotes that for process $P \sqcap Q$, the selection of process P or Q is controlled by the environment, i.e., the choice is resolved by the occurrence of the first visible event.

$$P \sqcap Q \triangleq \mathbf{H}((P \wedge Q) \triangleleft \text{Stop} \triangleright (P \vee Q))$$

The above definition shows that if no observation has been made and termination has not occurred (i.e., process *Stop* is true), then the process has both possible behaviours of P and Q . Alternatively, if an observation had been made, (i.e., process *Stop* is false.), then process behaviour will be either that of P or that of Q depending on from which choice is made.

4.1.3.7 State Guard

Process $[b]P$ waits until condition b becomes true and then behaves as P . Moreover, the checking of condition b is performed simultaneously with the occurrence of the first event of process P .

$$[b]P \triangleq P \triangleleft (\mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \triangleright \text{Stop}$$

The semantics states that if the Boolean guard b is satisfied under the state from the initial observation of P , represented by $\pi_1(\text{head}(tr' - tr))$, then the observation of whole process is the same as P , or otherwise, process behaves as process *Stop*. Function π_1 selects the first element of a tuple and function *head* returns the first element of a sequence. Note that the semantics of traditional conditional choice *if* (b) $\{P\}$ *else* $\{Q\}$ can be equivalent to the semantics of $[b]P \vee [\neg b]Q$.

4.1.3.8 Parallel Composition

The parallel composition $P \parallel Q$ executes P and Q in the following way: (1) common actions of P and Q require simultaneous participation, (2) synchronous channel output in

one process occurs simultaneously with the corresponding channel input in the other process, and (3) other events of processes occur independently.

In CSP, the semantics of parallel composition is defined in terms of the merge operator \parallel_M in UTP [31], where the predicate M captures how to merge two observations. To deal with channel-based communications and shared variable updates in CSP#, we here define a new merge predicate $M(X)$ to model the merge operation. The set X contains common actions of both processes (denoted by set X_1) and all synchronous channel inputs and outputs (denoted by set X_2). Namely,

$$P \parallel Q \triangleq \left(\begin{array}{l} P[0.ok, 0.wait, 0.ref, 0.tr/ok', wait', ref', tr'] \wedge \\ Q[1.ok, 1.wait, 1.ref, 1.tr/ok', wait', ref', tr'] \end{array} \right); M(X)$$

where

$$M(X) \triangleq \left(\begin{array}{l} (ok' = 0.ok \wedge 1.ok) \wedge \\ (wait' = 0.wait \vee 1.wait) \wedge \\ (ref' = (0.ref \cap 1.ref \cap X_2) \cup ((0.ref \cup 1.ref) \cap X_1) \\ \quad \cup ((0.ref \cap 1.ref) - X_1 - X_2)) \\ (tr' - tr \in (0.tr - tr \parallel_X 1.tr - tr)) \end{array} \right); Skip$$

The predicate $M(X)$ captures four kinds of behaviours of a parallel composition. First, the composition diverges if either process diverges (represented by predicate $ok' = 0.ok \wedge 1.ok$). Second, the composition terminates if both processes terminate ($wait' = 0.wait \vee 1.wait$). Third, the composition refuses synchronous channel outputs/inputs that are refused by both processes ($0.ref \cap 1.ref \cap X_2$), all actions that are in the set X_1 and refused by either process ($((0.ref \cup 1.ref) \cap X_1)$), and actions that are not in the set X_1 but refused by both processes ($((0.ref \cap 1.ref) - X_1 - X_2)$). Last, the trace of the composition is a member of the set of traces produced by the *trace synchronisation* function \parallel_X as elaborated below.

Function \parallel_X models how to merge two individual traces into a set of all possible traces; there are nine cases from six groups. In the following definitions, $s, s', s_1, s'_1, s_2, s'_2$ are representative elements of variable states, a, a_1, a_2 are representative elements of actions, ch is a representative element of channel names, and v is a value with type T .

- (1) If both input traces are empty, the result is a set of an empty sequence (denoted by **case-1**); (2) if only one input trace is empty, the result is determined based on the first observation of that non-empty trace: (i) if that observation is an action in the set X which requires synchronisation, then the result is a set containing only an empty sequence, or otherwise, the first observation is recorded in the merged trace (**case-2**); if the first observation is (ii) a channel input/output/communication (**case-3**) or (iii) a state pair (**case-4**), then the observation is recorded in the merged trace.

$$\textbf{case-1} \quad \langle \rangle \parallel_X \langle \rangle = \{\langle \rangle\}$$

$$\textbf{case-2} \quad \langle (s, a) \rangle \frown t \parallel_X \langle \rangle = \begin{cases} \{\langle \rangle\} & \text{if } a \in X \\ \{\langle (s, a) \rangle \frown l \mid l \in t \parallel_X \langle \rangle\} & \text{otherwise} \end{cases}$$

$$\textbf{case-3} \quad \langle (s, h) \rangle \frown t \parallel_X \langle \rangle = \{\langle (s, h) \rangle \frown l \mid l \in t \parallel_X \langle \rangle\}, \text{ where } h \in \{ch?v, ch!v, ch.v\}$$

$$\textbf{case-4} \quad \langle (s, s') \rangle \frown t \parallel_X \langle \rangle = \{\langle (s, s') \rangle \frown l \mid l \in t \parallel_X \langle \rangle\}$$

- When a communication is over a synchronous channel, if the first observations of two input traces match (see Definition 4 below), then a synchronisation may occur (denoted by the set \mathcal{G}_1) or at this moment a synchronisation does not occur (denoted by the set \mathcal{G}_2). Otherwise, a synchronisation cannot occur. Here, two observations are matched provided that both channel input and output from two processes respectively are enabled under the same pre-state.

Definition 4 (Match). *Given two pairs $p_1 = (s_1, h_1)$ and $p_2 = (s_2, h_2)$, we say that they are matched if both $s_1 = s_2$ and $\{h_1, h_2\} = \{ch?v, ch!v\}$ are satisfied, denoted as*

$\text{match}(p_1, p_2)$.

$$\text{case-5 } \langle (s_1, h_1) \rangle \frown t_1 \parallel_X \langle (s_2, h_2) \rangle \frown t_2 = \begin{cases} \mathcal{G}_1 \cup \mathcal{G}_2 & \text{match}((s_1, h_1), (s_2, h_2)) \\ \mathcal{G}_2 & \text{otherwise} \end{cases}$$

where $h_1, h_2 \in \{ch?v, ch!v, ch.v\}$, $\mathcal{G}_1 \hat{=} \{\langle (s_1, ch.v) \rangle \frown l \mid l \in t_1 \parallel_X t_2\}$, and $\mathcal{G}_2 \hat{=} \{\langle (s_1, h_1) \rangle \frown l \mid l \in t_1 \parallel_X \langle (s_2, h_2) \rangle \frown t_2\} \cup \{\langle (s_2, h_2) \rangle \frown l \mid l \in \langle (s_1, h_1) \rangle \frown t_1 \parallel_X t_2\}$.

- When two actions (a_1 and a_2) are synchronised, there are five cases with respect to the initial states (s_1 and s_2) and actions from the first observation of two input traces: (1) both actions are in the set X but different, (2) actions from X are the same but under different pre-states, (3) actions from X are the same and under the same pre-state, (4) one of the actions is not in X , and (5) both actions are not in X . As shown in **case-6** below, the result is a set containing only an empty sequence for cases (1) and (2). A synchronisation occurs under case (3), although it is postponed to occur under case (4). Either action can occur for case (5).

$$\text{case-6 } \langle (s_1, a_1) \rangle \frown t_1 \parallel_X \langle (s_2, a_2) \rangle \frown t_2 = \begin{cases} \{\langle \rangle\} & a_1, a_2 \in X \wedge a_1 \neq a_2 \\ \{\langle \rangle\} & a_1, a_2 \in X \wedge a_1 = a_2 \wedge s_1 \neq s_2 \\ \{\langle (s_1, a_1) \rangle \frown l \mid l \in t_1 \parallel_X t_2\} & a_1, a_2 \in X \wedge a_1 = a_2 \wedge s_1 = s_2 \\ \{\langle (s_1, a_1) \rangle \frown l \mid l \in t_1 \parallel_X \langle (s_2, a_2) \rangle \frown t_2\} & a_1 \notin X \wedge a_2 \in X \\ \{\langle (s_1, a_1) \rangle \frown l \mid l \in t_1 \parallel_X \langle (s_2, a_2) \rangle \frown t_2\} \\ \cup & a_1 \notin X \wedge a_2 \notin X \\ \{\langle (s_2, a_2) \rangle \frown l \mid l \in \langle (s_1, a_1) \rangle \frown t_1 \parallel_X t_2\} \end{cases}$$

- When the merge operation is on an action a and channel input $ch?v$, output $ch!v$, communication $ch.v$, or a post-state s'_2 , (1) if a is from the set X , then its occurrence is postponed (\mathcal{G}_3), (2) or otherwise, either observation from two processes occurs ($\mathcal{G}_3 \cup \mathcal{G}_4$).

$$\text{case-7 } \langle (s_1, a) \rangle \frown t_1 \parallel_X \langle (s_2, h) \rangle \frown t_2 = \begin{cases} \mathcal{G}_3 & \text{if } a \in X \\ \mathcal{G}_3 \cup \mathcal{G}_4 & \text{otherwise} \end{cases}$$

where $h \in \{ch?v, ch!v, ch.v, s'_2\}$, $\mathcal{G}_3 \triangleq \{\langle(s_2, h)\rangle \wedge l \mid l \in \langle(s_1, a)\rangle \wedge t_1 \parallel_X t_2\}$, and $\mathcal{G}_4 \triangleq \{\langle(s_1, a)\rangle \wedge l \mid l \in t_1 \parallel_X \langle(s_2, h)\rangle \wedge t_2\}$.

- When the merge operation is over two state pairs or the operation is on a state pair and a channel input/output/communication, either observation from two processes can occur as only one process can update shared variable(s) at a time when processes run in parallel.

case-8 $\langle(s_1, s'_1)\rangle \wedge t_1 \parallel_X \langle(s_2, h)\rangle \wedge t_2 = \{\langle(s_1, s'_1)\rangle \wedge l \mid l \in t_1 \parallel_X \langle(s_2, h)\rangle \wedge t_2\} \cup \{\langle(s_2, h)\rangle \wedge l \mid l \in \langle(s_1, s'_1)\rangle \wedge t_1 \parallel_X t_2\}$ where $h \in \{s'_2, ch?v, ch!v, ch.v\}$

- Finally, function \parallel_X is symmetric.

case-9 $t_1 \parallel_X t_2 = t_2 \parallel_X t_1$

4.1.3.9 Interleave

In the open environment, processes P and Q run independently (except communications through synchronous channels) for $P \parallel\parallel Q$. The semantics of the interleave operator defined below is similar to that of parallel operator except the set X which only contains synchronous channel outputs and inputs.

$$P \parallel\parallel Q \triangleq P \parallel_{M(X)} Q$$

The merge predicate $M(X)$ is the same as the definition in Section 4.1.3.8.

4.1.3.10 Hiding

The hiding operator makes all occurrences of actions in X_1 hidden from the environment of the process. The actions in set X_1 are not recorded in the process trace.

$$P \setminus X_1 \triangleq \mathbf{H}(\exists s \bullet P[s, X_1 \cup ref'/tr', ref'] \wedge (tr' - tr) = hide(s - tr, X_1)); Skip$$

The definition of hiding is defined by renaming the final trace of P as s , and restricting s to the trace which contains all the events of process P except those in set X_1 , captured by the function *hide*. The final refusal set is the union of refusal set of P and set X_1 . Note that *Action* denotes a set of actions.

$$\begin{aligned} \text{hide} &: \text{seq}((S \times E) \cup (S \times S^\perp)) \times \mathbb{P} \text{Action} \rightarrow \text{seq}((S \times E) \cup (S \times S^\perp)) \\ \text{hide}(\langle \rangle) &\hat{=} \langle \rangle \\ \text{hide}(\langle (s, e) \rangle \frown t) &\hat{=} \begin{cases} \text{hide}(t, X_1) & \text{if } e \in X_1 \\ \langle (s, e) \rangle \frown \text{hide}(t, X_1) & \text{otherwise} \end{cases} \end{aligned}$$

4.1.3.11 Recursion

Let X be a variable standing for a call to a recursive process, F be a monotonic function from CSP# processes to CSP# processes, the semantics of recursion $\mu X \bullet F(X)$ is defined as the weakest fixed point, which is the greatest lower bound of all the fixed points of F with the bottom element $H(\text{true})$ and the top element $H(\text{false})$; namely, $\sqcap \{X \mid X \sqsupseteq F(X)\}$. The definition of refinement order \sqsubseteq is shown in Definition 5.

4.1.3.12 Refinement

Refinement calculus is designed to produce correct programs, assisting in the software development. In the UTP theory, it is expressed as logic implication; an implementation (denoted as predicate P) satisfying a specification (denoted as predicate S) is formally expressed by universal quantification implication $\forall a, a', \dots \bullet P \Rightarrow Q$, where a, a', \dots are all observational variables of the alphabet, which must be the same for the specification and implementation. The universal quantification implication is usually denoted as $[P \Rightarrow Q]$. The definition of refinement in CSP# is given as below.

Definition 5 (Refinement). *Let P and Q be predicates for processes with the same shared*

variable state space, the refinement $P \sqsubseteq Q$ holds iff $[P \Rightarrow Q]$.

The refinement ordering in our definition is strong; every observation that satisfies P must also satisfy Q . The observation includes all process behaviours, i.e., stability, termination, traces, and refusals. Moreover, the record of the trace considers both variable states and event occurrences. For example, given a process $P = [x = 2]b \rightarrow \text{Skip} \sqcap [x \neq 2]c \rightarrow \text{Skip}$, and a process $Q = [x = 2]b \rightarrow \text{Skip} \sqcap [x \neq 2]d \rightarrow \text{Skip}$, the refinement $P \sqsubseteq Q$ does not hold although one observation satisfies both processes when x is equal to 2. A counterexample is that when x is not equal to 2, processes P and Q perform action c and d , respectively.

Notice that we only allow that in the trace sequence of process P , every element shall be the same as its counterpart in Q . In other words, our refinement prevents atomic program operations updating shared variables from being refined by non-atomic program operations which make the same effect. For example, given a process $P = e\{x = x + 1\} \rightarrow e\{x = x + 1\} \rightarrow \text{Skip}$, and a process $Q = e\{x = x + 2\} \rightarrow \text{Skip}$, the refinement $P \sqsubseteq Q$ does not hold.

Definition 6 (Equivalence). *For any two CSP# processes P and Q , P is equivalent to Q if and only if $P \sqsubseteq Q \wedge Q \sqsubseteq P$.*

Lemma 1 (Monotonicity). *All process combinators defined in the CSP# language are monotonic.*

The proofs of Lemma 1 are in Appendix B.

Theorem 1 (Compositional). *The open semantics of CSP# is compositional.*

Proof Given process combinator F and processes P, Q such that P and Q are equivalent with respect to the open semantics, we have $P \sqsubseteq Q$ and $Q \sqsubseteq P$ according to Definition 6. According to Lemma 1, both $F(P) \sqsubseteq F(Q)$ and $F(Q) \sqsubseteq F(P)$, which indicates $F(P) = F(Q)$, i.e., the open semantics is compositional. \square

4.2 Algebraic Laws

In this section, we present a set of algebraic laws concerning the distinct features of CSP#. All algebraic laws can be established based on our denotational model. That is to say, if the equality of two syntactically different processes is algebraically provable, then the two processes are also equivalent with respect to the denotational semantics. Moreover, these algebraic laws can be used as auxiliary reasoning rules to prove process equivalence during theorem proving.

4.2.1 State Guard

Law **guard - 1** enables the elimination of nested guards.

$$\mathbf{guard - 1} \quad [b_1]([b_2]P) = [b_1 \wedge b_2]P$$

Proof:

$$\begin{aligned}
& [b_1]([b_2]P) \tag{4.1.3.7} \\
&= \left(\begin{array}{c} (P \triangleleft (\mathcal{B}(b_2)(\pi_1(head(tr' - tr))) = true \wedge tr < tr') \triangleright Stop) \\ \triangleleft (\mathcal{B}(b_1)(\pi_1(head(tr' - tr))) = true \wedge tr < tr') \triangleright \\ Stop \end{array} \right) \tag{predicate calculus} \\
&= \left(\begin{array}{c} \left(\begin{array}{c} P \wedge \mathcal{B}(b_2)(\pi_1(head(tr' - tr))) = true \wedge \\ tr < tr' \wedge \\ \mathcal{B}(b_1)(\pi_1(head(tr' - tr))) = true \end{array} \right) \\ \vee \\ \left(\begin{array}{c} Stop \wedge \neg(\mathcal{B}(b_2)(\pi_1(head(tr' - tr))) = true \wedge \\ tr < tr' \wedge \\ \mathcal{B}(b_1)(\pi_1(head(tr' - tr))) = true \end{array} \right) \end{array} \right) \tag{Def. 2}
\end{aligned}$$

$$\begin{aligned}
& (P \wedge \mathcal{B}(b_2 \wedge b_1)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \\
= & \quad \vee \quad [4.1.3.7] \\
& (Stop \wedge \neg(\mathcal{B}(b_2 \wedge b_1)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr')) \\
= & [b_1 \wedge b_2]P \quad \square
\end{aligned}$$

Law **guard - 2** shows the distribution of the state guard through parallel composition, external choice and internal choice.

$$\textbf{guard - 2} \quad [b](P_1 \text{ op } P_2) = [b]P_1 \text{ op } [b]P_2 \quad \text{where, op} \in \{\parallel, \square, \sqcap\}$$

Proof: The guard b_1 constrains that the pre-state of the initial observation of the composition process should satisfy the condition, since the pre-state of the initial observation of the composition process can be from either process P_1 or P_2 (see Section 4.1.3.6, 4.1.3.8), so the condition should be satisfied by the initial observation of both processes. \square

Law **guard - 3** shows that process $[false]P$ behaves like $Stop$ because its guard can never be fired.

$$\textbf{guard - 3} \quad [false]P = Stop$$

Proof:

$$\begin{aligned}
& [false]P \quad [4.1.3.7] \\
= & P \triangleleft (\mathcal{B}(false)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \triangleright Stop \quad [Def. 2] \\
= & P \triangleleft (false \wedge tr < tr') \triangleright Stop \quad [predicate \text{ calculus}] \\
= & Stop \quad \square
\end{aligned}$$

4.2.2 Sequential Composition

$$\textbf{seq - 1} \quad (P_1; P_2); P_3 = P_1; (P_2; P_3)$$

$$\textbf{seq - 2} \quad P_1; (P_2 \sqcap P_3) = (P_1; P_2) \sqcap (P_1; P_3)$$

$$\textbf{seq - 3} \quad (P_1 \sqcap P_2); P_3 = (P_1; P_3) \sqcap (P_2; P_3)$$

seq - 4 $P = \text{Skip}; P$

seq - 5 $P = P; \text{Skip}$

seq - 1 shows that sequential composition is associative. **seq - 2, 3** show the distribution of sequential composition through external choice. **seq - 4, 5** show that process *Skip* is the left and right unit of sequential composition, respectively. The semantics of $\text{CSP}\#$ sequential composition and *Skip* is the same as in CSP, so the proofs of the above laws are not shown.

4.2.3 Parallel Composition

par - 1, 2 show that parallel composition is commutative and associative. Consequently, the order of parallel composition is irrelevant.

par - 1 $P_1 \parallel P_2 = P_2 \parallel P_1$

Proof:

$$\begin{aligned}
 & P_1 \parallel P_2 && [4.1.3.8] \\
 = & (P_1[0.\text{obs}/\text{obs'}] \wedge P_2[1.\text{obs}/\text{obs'}]); M(X) && \left[\begin{array}{l} \text{symmetry of } M(X) \text{ and} \\ \text{predicate calculus} \end{array} \right] \\
 = & (P_2[0.\text{obs}/\text{obs'}] \wedge P_1[1.\text{obs}/\text{obs'}]); M(X) && [4.1.3.8] \\
 = & P_2 \parallel P_1 && \square
 \end{aligned}$$

par - 2 $(P_1 \parallel P_2) \parallel P_3 = P_1 \parallel (P_2 \parallel P_3)$, provided that common actions and synchronous outputs/inputs among processes P_1 , P_2 and P_3 (denoted as set X) are the same.

Proof:

$$\begin{aligned}
& (P_1 \parallel P_2) \parallel P_3 && [4.1.3.8] \\
= & \left(\begin{array}{l} ((P_1[0.obs/obs'] \wedge P_2[1.obs/obs'])); \\ M(X)[0.obs/obs'] \\ \wedge \\ P_3[1.obs/obs'] \end{array} \right); M(X) && \left[\begin{array}{l} \text{associativity of } M(X), \\ \text{predicate calculus} \end{array} \right] \\
= & \left(\begin{array}{l} P_1[0.obs/obs'] \\ \wedge \\ ((P_2[0.obs/obs'] \wedge P_3[1.obs/obs'])); \\ M(X)[1.obs/obs'] \end{array} \right); M(X) && [4.1.3.8] \\
= & P_1 \parallel (P_2 \parallel P_3) && \square
\end{aligned}$$

Law **par - 3** shows that process *Skip* is the unit of parallelism.

par - 3 $Skip \parallel P = P = P \parallel Skip$

Proof:

$$\begin{aligned}
& Skip \parallel P && [\mathbf{par} - 1] \\
= & P \parallel Skip && \square
\end{aligned}$$

$$\begin{aligned}
& P \parallel Skip && [4.1.3.8] \\
= & P \parallel_{M(X)} Skip && [UTP \text{ parallel}] \\
= & ((P[0.obs/obs']) \wedge (Skip[1.obs/obs'])); M(X_1) && [4.1.3.1] \\
= & ((P[0.obs/obs']) \wedge (\mathbf{H}(\exists ref \bullet II)[1.obs/obs'])); M(X_1) && [\mathbf{H}]
\end{aligned}$$

$$\begin{aligned}
&= \left(\left((P[0.obs/obs']) \wedge \right. \right. \\
&\quad \left(\begin{array}{l} (wait \wedge \neg ok \wedge tr \leq tr') \vee \\ (wait \wedge ok' \wedge tr' = tr \\ \wedge wait' = wait \wedge ref' = ref) \vee \\ (\neg wait \wedge \neg ok \wedge tr \leq tr') \vee \\ (\neg wait \wedge \exists ref \bullet (ok' \wedge tr' = tr \wedge \\ wait' = wait \wedge ref' = ref)) \end{array} \right) \left. \begin{array}{l} \left[\begin{array}{l} 1.obs \\ /obs' \end{array} \right] \end{array} \right) ; M(X_1) \quad \left[\begin{array}{l} propo- \\ sitional \\ calculus \end{array} \right] \\
&= \left(\left((P[0.obs/obs']) \wedge \right. \right. \\
&\quad \left(\begin{array}{l} (\neg ok \wedge tr \leq tr') \vee \\ (wait \wedge ok' \wedge tr' = tr \\ \wedge wait' = wait \wedge ref' = ref) \vee \\ (\neg wait \wedge ok' \wedge tr' = tr \\ \wedge wait' = wait) \end{array} \right) \left. \begin{array}{l} \left[1.obs/obs' \right] \end{array} \right) ; M(X_1) \quad \left[\begin{array}{l} P \text{ is} \\ \mathbf{CSP1} \end{array} \right] \\
&= \left(\left((\mathbf{CSP1}(P)[0.obs/obs']) \wedge \right. \right. \\
&\quad \left(\begin{array}{l} (\neg ok \wedge tr \leq tr') \vee \\ (wait \wedge ok' \wedge tr' = tr \\ \wedge wait' = wait \wedge ref' = ref) \vee \\ (\neg wait \wedge ok' \wedge tr' = tr \\ \wedge wait' = wait) \end{array} \right) \left. \begin{array}{l} \left[1.obs/obs' \right] \end{array} \right) ; M(X_1) \quad [\mathbf{CSP1}] \\
&= \left(\left((P \vee \neg ok \wedge tr \leq tr')[0.obs/obs'] \wedge \right. \right. \\
&\quad \left(\begin{array}{l} (\neg ok \wedge tr \leq tr') \vee \\ (wait \wedge ok' \wedge tr' = tr \\ \wedge wait' = wait \wedge ref' = ref) \vee \\ (\neg wait \wedge ok' \wedge tr' = tr \\ \wedge wait' = wait) \end{array} \right) \left. \begin{array}{l} \left[\begin{array}{l} 1.obs \\ /obs' \end{array} \right] \end{array} \right) ; M(X_1) \quad [P \text{ is } \mathbf{R3}]
\end{aligned}$$

$$\begin{aligned}
&= \left(\left((\mathbf{R3}(P) \vee \neg ok \wedge tr \leq tr')[0.obs/obs']) \wedge \right. \right. \\
&\quad \left(\begin{array}{l} (\neg ok \wedge tr \leq tr') \vee \\ (wait \wedge ok' \wedge tr' = tr \wedge wait' = wait \\ \wedge ref' = ref) \vee \\ (\neg wait \wedge ok' \wedge tr' = tr \wedge wait' = wait) \end{array} \right) \left[\begin{array}{l} 1.obs \\ /obs' \end{array} \right] \right) ; M(X_1) \quad [\mathbf{R3}] \\
\\
&= \left(\left(\begin{array}{l} (wait \wedge II \vee \\ \neg wait \wedge P \vee \\ \neg ok \wedge tr \leq tr') \end{array} \right) [0.obs/obs']) \right. \\
&\quad \wedge \\
&\quad \left(\begin{array}{l} (\neg ok \wedge tr \leq tr') \vee \\ (wait \wedge ok' \wedge tr' = tr \wedge wait' = wait \\ \wedge ref' = ref) \vee \\ (\neg wait \wedge ok' \wedge tr' = tr \wedge wait' = wait) \end{array} \right) \left[\begin{array}{l} 1.obs \\ /obs' \end{array} \right] \right) ; M(X_1) \\
&\hspace{15em} [II \text{ and propositional calculus}] \\
&= \left(\left(\begin{array}{l} (\neg ok \wedge tr \leq tr') \vee \\ (wait \wedge ok' \wedge tr' = tr \wedge wait' = wait \\ \wedge ref' = ref) \vee \\ (\neg wait \wedge P) \end{array} \right) [0.obs/obs']) \right. \\
&\quad \wedge \\
&\quad \left(\begin{array}{l} (\neg ok \wedge tr \leq tr') \vee \\ (wait \wedge ok' \wedge tr' = tr \wedge wait' = wait \\ \wedge ref' = ref) \vee \\ (\neg wait \wedge ok' \wedge tr' = tr \wedge wait' = wait) \end{array} \right) \left[\begin{array}{l} 1.obs \\ /obs' \end{array} \right] \right) ; M(X_1) \\
&\hspace{15em} [predicate calculus]
\end{aligned}$$

$$= \left(\left(\begin{array}{l} (\neg ok \wedge tr \leq 0.tr) \vee \\ (wait \wedge 0.ok \wedge 0.tr = tr \\ \wedge 0.wait = wait \wedge 0.ref = ref) \vee \\ (\neg wait \wedge P[0.obs/obs']) \end{array} \right) \wedge \left(\begin{array}{l} (\neg ok \wedge tr \leq 1.tr) \vee \\ (wait \wedge 1.ok \wedge 1.tr = tr \\ \wedge 1.wait = wait \wedge 1.ref = ref) \vee \\ (\neg wait \wedge 1.ok \wedge 1.tr = tr \\ \wedge 1.wait = wait) \end{array} \right) \right) ; M(X_1) \left[\begin{array}{l} \text{propositional} \\ \text{calculus} \end{array} \right]$$

$$= \left(\begin{array}{l} (\neg ok \wedge tr \leq 0.tr \wedge tr \leq 1.tr) \vee \\ (\neg ok \wedge tr \leq 0.tr \wedge wait \wedge 1.ok \wedge 1.tr = tr \\ \wedge 1.wait = wait \wedge 1.ref = ref) \vee \\ (\neg ok \wedge tr \leq 0.tr \wedge \neg wait \wedge 1.ok \wedge 1.tr = tr \wedge 1.wait = wait) \vee \\ (\neg ok \wedge tr \leq 1.tr \wedge wait \wedge 0.ok \\ \wedge 0.tr = tr \wedge 0.wait = wait \wedge 0.ref = ref) \vee \\ (wait \wedge 0.ok \wedge 1.ok \wedge 0.tr = 1.tr = tr \\ \wedge 0.wait = 1.wait = wait \wedge 0.ref = 1.ref = ref) \vee \\ (\neg ok \wedge tr \leq 1.tr \wedge \neg wait \wedge (P[0.obs/obs'])) \vee \\ (\neg wait \wedge 1.ok \wedge 1.tr = tr \wedge 1.wait = wait \wedge P[0.obs/obs']) \end{array} \right) ; M(X_1)$$

[4.1.3.8 and $M(X)$]

$$\begin{aligned}
& \left(\begin{array}{l}
(\neg ok \wedge tr \leq 0.tr \wedge tr \leq 1.tr) \vee \\
(\neg ok \wedge tr \leq 0.tr \wedge wait \wedge 1.ok \\
\quad \wedge 1.tr = tr \wedge 1.wait = wait \wedge 1.ref = ref) \vee \\
(\neg ok \wedge tr \leq 0.tr \wedge \neg wait \wedge 1.ok \\
\quad \wedge 1.tr = tr \wedge 1.wait = wait) \vee \\
(\neg ok \wedge tr \leq 1.tr \wedge wait \wedge 0.ok \\
\quad \wedge 0.tr = tr \wedge 0.wait = wait \wedge 0.ref = ref) \vee \\
(wait \wedge 0.ok \wedge 1.ok \wedge 0.tr = 1.tr = tr \\
\quad \wedge 0.wait = 1.wait = wait \wedge 0.ref = 1.ref = ref) \vee \\
(\neg ok \wedge tr \leq 1.tr \wedge \neg wait \wedge (P[0.obs/obs'])) \vee \\
(\neg wait \wedge 1.ok \wedge 1.tr = tr \wedge 1.wait = wait \wedge (P[0.obs/obs']))
\end{array} \right) ; \\
= & \left(\left(\begin{array}{l}
(ok' = 0.ok \wedge 1.ok) \wedge \\
(wait' = 0.wait \vee 1.wait) \wedge \\
(ref' = ((0.ref_a \cup 1.ref_a) \cap X_1) \cup ((0.ref_a \cap 1.ref_a) - X_1) \\
\quad \cup (0.ref_c \cap 1.ref_c \cap X_2)) \wedge \\
(tr' - tr \in (0.tr - tr \parallel_X 1.tr - tr))
\end{array} \right) ; Skip \right) \\
& [\text{seq} - 1]
\end{aligned}$$

$$\begin{aligned}
&= \left(\left(\begin{aligned} &(\neg ok \wedge tr \leq 0.tr \wedge tr \leq 1.tr) \vee \\ &(\neg ok \wedge tr \leq 0.tr \wedge wait \wedge 1.ok \\ &\quad \wedge 1.tr = tr \wedge 1.wait = wait \wedge 1.ref = ref) \vee \\ &(\neg ok \wedge tr \leq 0.tr \wedge \neg wait \wedge 1.ok \\ &\quad \wedge 1.tr = tr \wedge 1.wait = wait) \vee \\ &(\neg ok \wedge tr \leq 1.tr \wedge wait \wedge 0.ok \\ &\quad \wedge 0.tr = tr \wedge 0.wait = wait \wedge 0.ref = ref) \vee \\ &(wait \wedge 0.ok \wedge 1.ok \wedge 0.tr = 1.tr = tr \\ &\quad \wedge 0.wait = 1.wait = wait \wedge 0.ref = 1.ref = ref) \vee \\ &(\neg ok \wedge tr \leq 1.tr \wedge \neg wait \wedge (P[0.obs/obs'])) \vee \\ &(\neg wait \wedge 1.ok \wedge 1.tr = tr \wedge 1.wait = wait \wedge (P[0.obs/obs'])) \end{aligned} \right) ; \end{aligned} \right. \\
&\quad \left(\begin{aligned} &(ok' = 0.ok \wedge 1.ok) \wedge \\ &(wait' = 0.wait \vee 1.wait) \wedge \\ &(ref' = ((0.ref_a \cup 1.ref_a) \cap X_1) \cup ((0.ref_a \cap 1.ref_a) - X_1) \\ &\quad \cup (0.ref_c \cap 1.ref_c \cap X_2)) \wedge \\ &(tr' - tr \in (0.tr - tr \parallel_X 1.tr - tr)) \end{aligned} \right) ; Skip
\end{aligned}$$

[4.1.3.5]

$$\begin{aligned}
&= \left(\left(\begin{aligned}
&(\neg ok \wedge tr \leq 0.tr \wedge tr \leq 1.tr) \vee \\
&(\neg ok \wedge tr \leq 0.tr \wedge wait \wedge 1.ok \\
&\quad \wedge 1.tr = tr \wedge 1.wait = wait \wedge 1.ref = ref) \vee \\
&(\neg ok \wedge tr \leq 0.tr \wedge \neg wait \wedge 1.ok \\
&\quad \wedge 1.tr = tr \wedge 1.wait = wait) \vee \\
&(\neg ok \wedge tr \leq 1.tr \wedge wait \wedge 0.ok \\
&\quad \wedge 0.tr = tr \wedge 0.wait = wait \wedge 0.ref = ref) \vee \\
&(wait \wedge 0.ok \wedge 1.ok \wedge 0.tr = 1.tr = tr \\
&\quad \wedge 0.wait = 1.wait = wait \wedge 0.ref = 1.ref = ref) \vee \\
&(\neg ok \wedge tr \leq 1.tr \wedge \neg wait \wedge (P[0.obs/obs'])) \vee \\
&(\neg wait \wedge 1.ok \wedge 1.tr = tr \wedge 1.wait = wait \\
&\quad \wedge (P[0.obs/obs']))
\end{aligned} \right) \right. \\
&\quad \wedge \\
&\quad \left(\begin{aligned}
&(ok' = 0.ok \wedge 1.ok) \wedge \\
&(wait' = 0.wait \vee 1.wait) \wedge \\
&(ref' = ((0.ref_a \cup 1.ref_a) \cap X_1) \cup ((0.ref_a \cap 1.ref_a) - X_1) \\
&\quad \cup (0.ref_c \cap 1.ref_c \cap X_2)) \wedge \\
&(tr' - tr \in (0.tr - tr \parallel_X 1.tr - tr))
\end{aligned} \right)
\end{aligned} \right) ; Skip$$

[predicate calculus]

$$\begin{aligned}
&= \left(\begin{array}{l}
(\neg ok \wedge tr \leq tr') \vee \\
(\neg ok \wedge wait \wedge tr \leq tr' \wedge ok' = 0.ok \wedge \Psi_1) \vee \\
(\neg ok \wedge \neg wait \wedge tr \leq tr' \wedge wait' = 0.wait) \vee \\
(\neg ok \wedge wait \wedge tr \leq tr' \wedge ok' = 1.ok \wedge \Psi_2) \vee \\
(wait \wedge ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) \vee \\
(\neg ok \wedge \neg wait \wedge tr \leq tr' \wedge \Psi_3) \vee \\
(\neg wait \wedge ok' = 0.ok \wedge tr' = 0.tr \wedge wait' = 0.wait \\
\wedge ref' = 0.ref \wedge (P[0.obs/obs']))
\end{array} \right)^2; Skip \quad \left[\begin{array}{l} predicate \\ calculus \end{array} \right] \\
&= \left(\begin{array}{l}
(\neg ok \wedge tr \leq tr') \vee \\
(wait \wedge ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref) \vee \\
(\neg wait \wedge P)
\end{array} \right); Skip \quad [II] \\
&= \left(\begin{array}{l}
(wait \wedge II) \vee \\
(\neg wait \wedge P) \vee \\
(\neg ok \wedge tr \leq tr')
\end{array} \right); Skip \quad [\mathbf{R3}] \\
&= \left(\begin{array}{l}
\mathbf{R3}(P) \vee \\
(\neg ok \wedge tr \leq tr')
\end{array} \right); Skip \quad [P \text{ is } \mathbf{R3}] \\
&= (P \vee \neg ok \wedge tr \leq tr'); Skip \quad [\mathbf{CSP1}] \\
&= \mathbf{CSP1}(P); Skip \quad [P \text{ is } \mathbf{CSP1}] \\
&= P; Skip \quad [\mathbf{seq} - 5] \\
&= P \quad \square
\end{aligned}$$

² Ψ_1, Ψ_2 and Ψ_3 are logic formulae in terms of ref' , $0.ref$ and $1.ref$.

4.3 The Closed Semantics

So far, we have constructed an open semantics for $\text{CSP}\#$. Namely, the denotational semantics is defined in an open environment. The interference by the environment is implicitly captured in the hybrid trace which collects the potential events or state transitions in which a process may engage. For example, given a trace $\langle\langle s_1, s'_1 \rangle\rangle \wedge \langle\langle s_2, e \rangle\rangle$, the transition from state s'_1 to s_2 is implicit, and it is performed by the environment. In addition, the environment can change the states, so it is not necessary to ensure that state s'_1 is the same as s_2 . Thus the system and environment alternate in making transitions. From Theorem 1, the open semantics maintains the compositionality of the processes. Therefore, it supports compositional verification of process behaviours.

However, if we look at it in another light, there is no need to retain all possible transitions from the environment if we have already built the model of the whole system or the behaviour of the environment has been modelled as a process. In this situation, we attempt to consider a closed semantics for the $\text{CSP}\#$ language. Fortunately, the closed semantics does not need to be defined from the scratch; it can be generated from the open semantics. Thus, we first introduce the definition of closed traces to judge which trace exactly describes the process behaviour in a closed environment.

Definition 7 (Closed Trace). *A hybrid trace tr is closed, represented as $\text{cl}(tr)$, if it satisfies the following two conditions.*

- (1) *For any state pair which is not the last element in the trace, the post-state is passed as the pre-state of its immediate subsequent element, i.e., $\forall 0 \leq i < \#tr - 1, \exists s, s' \in \mathbf{S} \bullet (tr_i = (s, s') \Rightarrow s' = \pi_1(tr_{(i+1)}))$ ³.*
- (2) *For any event which is not the last element in the trace, it should share the same pre-state with its immediate subsequent element, i.e., $\forall 0 \leq i < \#tr - 1, \exists s \in \mathbf{S}, e \in \mathbf{E} \bullet (tr_i =$*

³ tr_i returns the $(i + 1)$ th element of the sequence tr .

$$(s, e) \Rightarrow s = \pi_1(tr_{(i+1)}).$$

Informally speaking, a closed trace has this property: two adjacent elements in the trace are associated by a common state; the post-state of the former equals to the pre-state of the latter if the former is a state transition; the pre-state is shared if the former is an event. Note that every element in a hybrid trace has a pre-state but only the state transition possesses a post-state because the pre-state is not changed when an event occurs. Since the environment cannot update the shared state, a closed trace is identified as the behaviour of the process in the closed environment. For convenience, given a set of hybrid traces, denoted as the set HT , we define $CL(HT)$ to represent the set of all closed traces in HT . Obviously, we have $CL(HT) \subseteq HT$.

Now, we can generate the closed semantics (denoted by $\llbracket P \rrbracket_{closed}$) from the open semantics ($\llbracket P \rrbracket_{open}$) for any communicating process P . The relation between them is revealed by Definition 8.

Definition 8 (Closed Semantics). $\llbracket P \rrbracket_{closed} \hat{=} \llbracket P \rrbracket_{open} \wedge cl(tr) \wedge cl(tr')$

According to the open semantics, two processes that are semantically equivalent can generate the same traces tr, tr' . Further, any two closed traces generated from their open traces are the same. Thus the equality with respect to the open semantics is preserved by the closed semantics, which is shown in Theorem 2.

Theorem 2. $\llbracket P \rrbracket_{open} = \llbracket Q \rrbracket_{open} \Rightarrow \llbracket P \rrbracket_{closed} = \llbracket Q \rrbracket_{closed}$

However, we cannot imply that $\llbracket P \rrbracket_{open} = \llbracket Q \rrbracket_{open}$ is true when $\llbracket P \rrbracket_{closed} = \llbracket Q \rrbracket_{closed}$ holds. Furthermore, given that $\llbracket P \rrbracket_{closed} = \llbracket Q \rrbracket_{closed}$, the law $P \parallel R = Q \parallel R$ may be invalid; the compositionality fails in the closed semantics as shown by Example 1.

Example 1. Given a process $P = a\{x = 2\} \rightarrow ([x = 2]b \rightarrow Skip \sqcap [x \neq 2]c \rightarrow Skip)$, and a process $Q = a\{x = 2\} \rightarrow ([x = 2]b \rightarrow Skip \sqcap [x \neq 2]d \rightarrow Skip)$, the closed semantics

of processes P and Q is the same, while their open semantics is not the same because after executing event a , process P may execute event c , and process Q may execute event d when the value of variable x is not equal to 2 in their pre-states. Therefore, given a process $R = e\{x = 3\} \rightarrow \text{Skip}$, there is a case that after executing the events a and e sequentially, process $P \parallel R$ will execute event c while process $Q \parallel R$ will execute event d , and thus the law $P \parallel R = Q \parallel R$ is not satisfied.

4.4 Summary

In this chapter, we have defined a denotational semantics in an open environment for the $\text{CSP}\#$ language based on the UTP framework. The formalised semantics covers different types of concurrency, i.e., communications and shared variable paradigm. In addition, a set of algebraic laws have been proposed based on the denotational model for $\text{CSP}\#$. Furthermore, a *closed* semantics has been derived from the open denotational semantics by focusing on the particular hybrid traces.

The work in this chapter is related to the denotational semantics of CSP which has been defined using two approaches. On one hand, Roscoe [62] and Hoare [29] provided a *trace* model, a *stable-failures* model and a *failures-divergences* model for CSP processes. In the trace model, every process is mapped to a set of traces which capture sequences of event occurrences during the process execution. In the stable-failures model, every process is mapped to a set of pairs, and each pair consists of a trace and a refusal. In the failures-divergences model, every process is mapped to a pair, where one component is a set of traces that can lead to divergent behaviours, and the other component contains all stable failures which are all pairs, and each pair is in the form of a trace and a refusal. On the other hand, Hoare and He [31] defined a denotational semantics for CSP processes using the UTP theory. Each process is formalised as a relation between an initial observation and

a subsequent observation; such relations are represented as predicates over observational variables which record process stability, termination, traces and refusals *before* or *after* the observation. Cavalcanti and Woodcock [14] related the UTP theory of CSP to the *failures-divergences* model of CSP.

The aforementioned denotational semantics for CSP does not deal with data aspects. To solve this problem, several attempts have been done to provide the denotational semantics for languages which integrate CSP with state-based notations. For example, Oliveira *et al.* [53] presented a denotational semantics for Circus based on a UTP theory. The proposed semantics includes two parts: one is for Circus actions, guarded commands, etc., and the other is for Circus processes which contain an encapsulated state, a main action, etc. However, this proposed semantics assumes that the sets of variables in processes shall be *disjoint* when those processes run in parallel or interleaving. Qin *et al.* [61] formalised the denotational semantics of Timed Communicating Object Z (TCOZ) based on the UTP framework. Their unified semantic model can deal with channel-based and sensor/actuator-based communications, although shared variables in TCOZ are restricted to only sensors/actuators.

There exists some work on shared-variable concurrency. Brooks [10] defined a denotational semantics for a shared-variable parallel language, where the semantic model considers state transitions only, and thus cannot be directly applied to communicating processes. Zhu *et al.* [97] derived a denotational semantics from the proposed operational semantics for the hardware description language Verilog. In addition, they [98] derived the denotational semantics from the algebraic semantics for Verilog to explore the equivalence of two semantic models. Recently, they [100] proposed a probabilistic language *PTSC* which integrates probability, time and shared-variable concurrency. The operational semantics of PTSC is explored and a set of algebraic laws are presented via bisimulation. Furthermore, a denotational semantics using the UTP approach [99] is derived from the algebraic laws based on the head normal form of PTSC constructs. These semantic models lack expressive

power to capture more complicated system behaviours like channel-based communications.

Chapter 5

Encoding CSP# Denotational Semantics into PVS

There are two popular approaches in formal verification, model checking [16, 5] and theorem proving [15, 69]. The model checking approach constructs a finite model of a system and checks automatically and exhaustively if this model meets a property; a great advantage of this approach is the fully automatic checking process, although the exhaustive state exploration feature constrains its support for infinite state systems in general. As a complementary approach to model checking, the theorem proving approach consists of specifying both systems and properties in certain mathematical logic and proving the correctness by logic inference; the formal proofs are often developed with interactive theorem provers.

In Chapter 4, we have defined a denotational semantics for CSP# in order to complement the existing CSP# operational semantics which was used for the model checking approach for CSP#. In addition, we also presented a set of algebraic laws and proved the soundness of these laws with respect to the denotational semantics. However, manually proving such laws is difficult and tedious, and subtle mistakes or omissions can easily occur at any stage of

the proofs. Moreover, a high grade of automated verification of system properties can save much human effort. Therefore, a tool that allows semantics mechanisation and supports mechanical proofs is needed.

In this chapter, we mechanise the CSP# denotational semantics in the Prototype Verification System (PVS) [54], which is a fundamental and the most important step towards the theorem proving approach for CSP#. PVS is an interactive theorem prover based on classical higher-order logic, similar to other theorem provers such as HOL [26], Isabelle [58] and Coq [6]. We use PVS as a demonstration of our encoding. Certainly we can also choose other powerful theorem provers like Coq to achieve the goal of mechanising CSP# semantics.

Mechanisation is also known as semantic embedding which encodes one specification language or logic into another. Semantic embedding is usually classified into *deep* embedding and *shallow* embedding [8, 3, 4]. The former formalises both syntax and semantics of the embedded language inside the host language, and the latter concentrates only the semantics of the embedded language [93]. We apply a hybrid embedding [28] to CSP#, namely, deep embedding of expressions and sequential programs and shallow embedding of process semantics. Our approach can avoid complex and tedious effort on defining abstract datatypes and semantic functions for the expressive CSP# process operators. Nevertheless, theorems about CSP# process semantics can still be proved as shown below.

Our mechanisation approach includes three parts which will be illustrated in the following subsections. First the theory of semantic model defines observational variables and healthiness conditions (Section 5.1). Based on the semantic model theory, we define the theory of expressions and programs which encodes carefully the syntax and semantics of arithmetic expressions, Boolean expressions and sequential programs (Section 5.2). Further, the semantics of processes and refinement relationship are formalised in the theory of process (Section 5.3). At the end of this section, we conduct mechanical proofs of important algebraic laws and lemmas based on the encoding in PVS (Section 5.4).

5.1 The Theory of Semantic Model

The first challenge of the semantic model encoding is to develop an appropriate data structure to represent the observational variables and relations over observational variables. The second challenge is to capture different types of events dedicated to CSP#. To address these challenges, we adopt PVS abstract datatype constructor to handle event types and PVS set theory to model relations as illustrated below.

5.1.1 The Theory of Observational Variables

CSP# supports concurrency over communications and shared variables. We first define the shared variable state and event type in PVS. In Chapter 4, Section 4.1.2, we use T to denote the abstract type for variable values, and here we initialise the type to be `int` which is the type of integer in PVS for simplicity. The following shows the formalised type for variable states (`S`), the non-terminating state (`abort`) and all states (`S_abort`).

```
Vars: TYPE+
S: TYPE+ = [Vars -> int]
abort: TYPE+ = {bot}
S_abort: DATATYPE
BEGIN
  is_S(left_s: S): is_S?
  is_abort(right_b: abort): is_abort?
END S_abort
```

In the above specification, type `S` is encoded as a function from variable type `Vars` to `int`. We define all states as a disjoint union by using a PVS abstract datatype: functions `is_S` and `is_abort` are constructors, functions `is_S?` and `is_abort?` are recognisers of the type `[S_abort -> bool]`, and determine whether the argument is constructed using the corresponding constructor. Note that a similar fashion of applying the PVS abstract datatype is

used throughout the rest of the section to model complex $\text{CSP}\#$ types.

$\text{CSP}\#$ supports both event synchronisation and pairwise handshake through synchronous channels. Thus event E includes actions, synchronous channel inputs, outputs and communications. To represent E , we define a datatype as follows.

```

E: DATATYPE WITH SUBTYPES RefE, Channelcom
BEGIN
  action(ac:Ta): action?: RefE
  input(ci:Ti): input?: RefE
  output(co:To): output?: RefE
  chancom(cm:Tm): chancom?: Channelcom
END E

```

Here, Ta is the type of actions, Ti is the type of channel inputs, To is the type of channel outputs, and Tm is the type of channel communications. Subtypes **RefE** and **Channelcom** denote the set of refused events and channel communications, respectively.

In our semantics, programs are interpreted as relations between initial observations and subsequent observations of their execution behaviours. Namely, relations are represented as predicates over observational variables. In Chapter 4, Section 4.1.1.1, we have defined eight variables to capture all aspects of program behaviours. In PVS, we use a record type to represent all observational variables, and a set of such records to represent a relation.

```

Obs: TYPE = [# ok:bool, ok1:bool, wait:bool, wait1:bool, ref:set[RefE], ref1:set[RefE],
              tr:Trace, tr1:Trace #]
Relation: TYPE = set[Obs]

```

In the above formalisation, a dashed variable is represented by its undashed variable name suffixed with number 1, e.g., $ok1$ denotes variable ok' . In our semantics for the trace, we use sequence to record the observations on the interaction of the process with its environment. We use the PVS predefined datatype `list` explained in Chapter 2, Section 2.3 to represent the sequence. Thus an empty sequence can be represented by a null list (`null`), and sequence

Predicate	PVS
P	$\{\text{pre:Obs} \mid P(\text{pre})\}$
$\neg P$	$\{\text{pre:Obs} \mid \text{NOT } P(\text{pre})\}$
$P \wedge Q$	$\{\text{pre:Obs} \mid P(\text{pre}) \text{ AND } Q(\text{pre})\}$
$P \vee Q$	$\{\text{pre:Obs} \mid P(\text{pre}) \text{ OR } Q(\text{pre})\}$
$P \Rightarrow Q$	$\{\text{pre:Obs} \mid P(\text{pre}) \text{ IMPLIES } Q(\text{pre})\}$

Table 5.1: Predicate formalisation in PVS

concatenation can be formalised by the predefined function **append** over the list.

```

StateTrans: TYPE+ = [S, S_abort]
StateEvent: TYPE+ = [S, E]
SE: DATATYPE
BEGIN
  s_t(s_sl:StateTrans): s_t?
  s_e(s_event:StateEvent): s_e?
END SE
Trace: TYPE = list[SE]

```

Based on the formalisation of observational variables and relation, we next illustrate how to formalise the detailed predicates in PVS. In general, a predicate P on the observational variables is encoded as a set $\{\text{pre:Obs} \mid P(\text{pre})\}$, and logic operators \neg , \wedge , \vee and \Rightarrow are formalised as **NOT**, **AND**, **OR**, and **IMPLIES** respectively in PVS. A summary of the formalisation is shown in Table 5.1, where P and Q are predicates, and **P** and **Q** are relations in PVS.

5.1.2 The Theory of Healthiness Conditions

$\text{CSP}\#$ satisfies the healthiness conditions **R1** to **R3** for reactive processes. In this section, we concentrate the encoding of **R1** and **R3** in PVS, and **R2** will be handled as one future step.

```

P: VAR Relation

R1(P): Relation = {pre:Obs|P(pre) AND pre'tr <= pre'tr1}

II: Relation = {pre:Obs|(pre'ok = false AND pre'tr <= pre'tr1) OR (pre'ok1 = true AND
    pre'tr1 = pre'tr AND pre'wait1 = pre'wait AND pre'ref1 = pre'ref)}

R3(P): Relation = {pre:Obs|IF pre'wait = true THEN II(pre) ELSE P(pre) ENDIF}

```

Healthiness condition **R1** is formalised as a function R1. Specifically R1 takes an arbitrary relation P as an input and returns a relation satisfying a predicate which is modelled as a set of observational variables (also called records); each record is a member of the relation (denoted as $P(\text{pre})$) and its final trace extends the initial trace (denoted as $\text{pre'tr} \leq \text{pre'tr1}$). Function R3 specifies that for each element pre , if the value of pre'wait is true, then pre is a member of the reactive identity relation II, or otherwise, it is a member of relation P.

In addition, $\text{CSP}\#$ satisfies two healthiness conditions **CSP1** and **CSP2** for communicating sequential processes, defined as follows.

```

CSP1(P): Relation = {pre: Obs|(pre'ok = false AND pre'tr <= pre'tr1) OR P(pre)}

CSP2(P): Relation = {pre:Obs|EXISTS (p:Obs): P(p) AND
    (pre'ok = p'ok AND pre'wait = p'wait AND pre'ref = p'ref AND pre'tr = p'tr) AND
    ((p'ok1 => pre'ok1) AND pre'wait1 = p'wait1 AND pre'ref1 = p'ref1 AND pre'tr1 = p'tr1)}

```

Function CSP1 denotes that for each element pre , when the value of pre'ok is false, its trace shall be extended, or it remains unchanged. The sequential composition ; in $\text{CSP2}(P) = P; (ok \Rightarrow ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref)$ is explicitly formalised; namely, for each element pre , its undashed variable value is the same as the undashed variable value of an element from relation P, e.g., $\text{pre'ok} = p'ok$, while its dashed variable value is the same as the dashed variable value from the second program, e.g., ok' is represented by pre'ok1 . Meanwhile, the dashed variable value of an element from relation P is the same as the undashed variable value in the second program, e.g., ok in the second program is represented by $p'ok1$.

Finally, processes in $\text{CSP}\#$ are defined by satisfying all the healthiness conditions. Our definition of processes relies on PVS subtyping: **process** is a subtype of **Relation**, where function **H** is the composition of four healthiness condition functions, and predicate $\text{H}(\text{P}) = \text{P}$ depicts that **P** is idempotent.

```
H(P): Relation = CSP2(CSP1(R3(R1(P))))
process: TYPE = {P | H(P) = P}
```

5.2 The Theories of Expressions and Programs

In this section, we first present the encoding of arithmetic expression syntax and semantics in PVS as below. Here, we define an abstract datatype **Aexp** to represent the syntax of arithmetic expressions, and a recursive function **aeval** to denote the semantics of arithmetic expressions, which uses PVS case construct to capture different expression types in **Aexp**.

```

% syntax for arithmetic expressions
Aexp: Datatype
BEGIN
  anum(n:int): anum?
  avar(x:Vars): avar?
  aplus(exp1,exp2:Aexp): apluse?
  aminus(exp1,exp2:Aexp): aminus?
  amult(exp1,exp2:Aexp): amult?
END Aexp

% auxiliary type for arithmetic expressions
S_int: TYPE+ = [S -> int]

% semantics for arithmetic expressions
aeval(a: Aexp): RECURSIVE S_int =
  (CASES a of
    anum(n): lambda (s:S): n,
    avar(x): lambda (s:S): s(x),
    aplus(exp1,exp2): lambda (s:S): (aeval(exp1)(s) + aeval(exp2)(s)),
    aminus(exp1,exp2): lambda (s:S): (aeval(exp1)(s) - aeval(exp2)(s)),
    amult(exp1, exp2): lambda (s:S): (aeval(exp1)(s) * aeval(exp2)(s))
  ENDCASES)
MEASURE a by <<

```

The formalisation of the syntax and semantics of Boolean expressions is similar to that of arithmetic expressions, shown below.

```

% syntax for boolean expressions
Bexp: Datatype
BEGIN
  bbool(b:bool): bbool?
  beq(exp1,exp2:Aexp): beq?
  blt(exp1,exp2:Aexp): blt?
  bnot(b:bool): bnot?
  band(b1,b2:Bexp): band?
  bor(b1,b2:Bexp): bor?
END Bexp

% auxiliary type for arithmetic expressions
S_bool: TYPE+ = [S -> bool]

% semantics for boolean expressions
beval(b: Bexp): RECURSIVE S_bool =
  (CASES b of
    bbool(b): lambda (s:S): b,
    beq(exp1,exp2): lambda (s:S):
      (IF aeval(exp1)(s) = aeval(exp2)(s) THEN TRUE ELSE FALSE ENDIF),
    blt(exp1,exp2): lambda (s:S):
      (IF aeval(exp1)(s) < aeval(exp2)(s) THEN TRUE ELSE FALSE ENDIF),
    bnot(b): lambda (s:S): (NOT b),
    band(b1,b2): lambda (s:S): (beval(b1)(s) AND beval(b2)(s)),
    bor(b1,b2): lambda (s:S): (beval(b1)(s) OR beval(b2)(s))
  ENDCASES)
MEASURE b BY <<

```

Base on the formalisation of arithmetic and Boolean expressions, we present the encoding of program syntax and semantics below.

```

1  % program syntax
2  Prog: Datatype
3  BEGIN
4    skip: skip?
5    assign(x:Vars, exp:Aexp): assign?
6    seq(prog1,prog2: Prog): seq?
7    if_prog(ifcond:Bexp, thn,els:Prog): if?
8  END Prog
9
10 % state relation
11 S_S_abort: TYPE+ = [S -> S_abort]
12 % skip function
13 skip_ceval:S_S_abort = lambda (s:S): is_S(s)
14 % update function: ignore checking the variable declaration
15 update_ceval (x:Vars, v: S_int): S_S_abort =
16   lambda (s: S): (let n = v(s) in is_S(s WITH [(x) := n]))
17 % sequence function
18 seq_ceval(s_prog1, s_prog2: S_S_abort): S_S_abort =
19   lambda (s:S): (IF is_abort?(s_prog1(s)) THEN is_abort(bot)
20     ELSE let s0 = s_prog1(s) in s_prog2(left_s(s0)) ENDIF)
21 % if function
22 if_ceval(s_b:S_bool, s_prog1, s_prog2: S_S_abort): S_S_abort =
23   lambda (s:S): (IF s_b(s) THEN s_prog1(s) ELSE s_prog2(s) ENDIF)
24
25 % sequential program semantics
26 ceval(p:Prog): RECURSIVE S_S_abort =
27   (CASES p of
28     skip: skip_ceval,
29     assign(x,exp): update_ceval (x, aeval(exp)),
30     seq(prog1,prog2): seq_ceval(ceval(prog1),ceval(prog2)),
31     if_prog(ifcond,thn,els): if_ceval(beval(ifcond),ceval(thn),ceval(els))
32   ENDCASES)
33 MEASURE p BY <<

```

In the above PVS specifications, we model the program semantics by a recursive function `ceval`, which uses PVS case construct to capture different program types in `Prog`. For example, the evaluation of sequential composition program (at line 30) is defined by a function `seq_ceval` which has two branches: given a state `s`, if the evaluation of program `s_prog1` does not terminate, then the function returns a state representing nontermination (at line 19), or otherwise, it returns the final state after the execution of two sequential programs (at line 20). Note that the type of `s0` is `S_abort`, if it is directly encoded as the input of `s_prog2`, the typechecking finds an error which is a wrong type of the first argument to `s_prog2`; namely, it expects type `S` but finds type `S_abort`, thus we use the accessor to cast the type to be `S` (denoted as `left_s(s0)`). Here, functions `aeval` at line 29 and `beval` at line 31 evaluate arithmetic and Boolean expressions respectively. Currently, while-loop program is not supported in our framework, and it will be one of our future works.

5.3 The Theory of Processes

PVS has a fixed syntax, and users cannot introduce new symbols. Thus we cannot directly use the standard $\text{CSP}\#$ process notations. Instead, we use the existing symbols in PVS, and summarize the standard $\text{CSP}\#$ syntax and our PVS encoding in Table 5.2, where $\mathbf{X1}$ is a set of actions of the type `Ta`, and process `Pi` is a parametric process of the type `[int -> process]`. Note that recursive processes cannot be directly defined in PVS. We apply ‘ μ -calculus’ theory [31] to formalising them, and the detailed formalisation is presented in Section 5.3.12.

5.3.1 Primitives

Following the guideline on how to formalise a relation in PVS (Section 5.1.1), it is straightforward to define primitive processes *Stop* and *Skip* in PVS, shown as follows.

Operation	CSP#	PVS
Stop	$Stop$	Stop
Skip	$Skip$	Skip
event prefixing	$a \rightarrow P$	$a \gg P$
channel output	$ch!exp \rightarrow P$	$ch_o_exp \gg P$
channel input	$ch?m \rightarrow P(m)$	$ch_i_m \gg P_i$
data operation prefixing	$e\{prog\} \rightarrow P$	$e_prog \gg P$
state guard	$[b]P$	$[](b,P)$
external choice	$P \square Q$	$P \wedge Q$
internal choice	$P \sqcap Q$	$P \vee Q$
sequential composition	$P; Q$	$P ++ Q$
hiding	$P \setminus X_1$	$Hid(P, X_1)$
parallel	$P \parallel Q$	$Par(P, Q)(X)$
interleaving	$P Q$	$Inter(P, Q)(X)$

Table 5.2: CSP# process syntax

```

Stop: process = H({pre|pre'ok1=true AND pre'tr1=pre'tr AND pre'wait1=true});
Skip: process = H({pre|(pre'ok=false AND pre'tr<=pre'tr1) OR
(pre'ok1=true AND pre'tr1=pre'tr AND pre'wait1=pre'wait)});

```

5.3.2 Sequential Composition

The semantics of sequential composition $P; Q$ is defined as the merge of two processes with the value of dashed observational variables in P being the same as the value of undashed variables in Q . The formalisation of sequential composition in PVS is defined as follows.

```

P, Q: VAR process
++(P, Q): process = H({pre|EXISTS (p,q: Obs): P(p) AND Q(q) AND
(pre'ok=p'ok AND pre'wait=p'wait AND pre'ref=p'ref AND pre'tr=p'tr) AND
(p'ok1=q'ok AND p'wait1=q'wait AND p'ref1=q'ref AND p'tr1=q'tr) AND
(pre'ok1=q'ok1 AND pre'wait1=q'wait1 AND pre'ref1=q'ref1 AND pre'tr1=q'tr1)});

```

To be specific, for each element **pre**, its undashed variable value is the same as the undashed variable value of an element from process P , and its dashed variable value is the same as the dashed variable value of an element from Q . Meanwhile, the dashed variable

value of an element from P is the same as the undashed variable value of an element from Q , e.g., $p'ok1=q'ok$.

5.3.3 Event prefixing

The formalisation of event prefixing $a \gg P$ contains two parts: first we define a prefixed action $\text{Skip}(a)$, and then compose the action with process P .

```

Skip(a): process = H({pre|pre'ok1=true AND
  ((pre'wait1=true AND NOT pre'ref1(action(a)) AND pre'tr1=pre'tr) OR
    (pre'wait1=false AND EXISTS(s:S): (pre'tr1=append(pre'tr,
      cons(s_e(s,action(a)),null))))));
>>(a, P): process = (Skip(a) ++ P);

```

Here the PVS predefined function `append` appends the action a to the end of the trace pre'tr . Constructor `cons` takes two arguments, the first is a pair `s_e(s,action(a))` consisting of a pre-state and an event, and the second is an empty list, and returns a trace capturing the behaviour of action a .

5.3.4 Synchronous Channel Output/Input

We first construct three tuples to respectively represent the type of synchronous channel input, output, and communication. Each tuple consists of three elements: the first is a string denoting a channel name, the second is a flag denoting the communication type, and the third is a number indicating the message.

```

% Type for input, output and communication symbol
T_i: TYPE+ = i; T_o: TYPE+ = o; T_m: TYPE+ = m
% Type for channel input, output and communication
Ti: TYPE = [string, T_i, int]
To: TYPE = [string, T_o, int]
Tm: TYPE = [string, T_m, int]

```

We next encode the syntax of channel input and output into PVS, similar to the way of defining above, where type **Vars** denotes variable names (Section 5.1.1) and type **Aexp** denotes the syntax of arithmetic expressions (Section 5.2).

```
% Syntax type for channel output/input/communication
Ti_syntax: TYPE = [string, T_i, Vars]
To_syntax: TYPE = [string, T_o, Aexp]
```

Based on the above encoding of event type and syntax definitions, the synchronous channel output is defined as follows.

```
1  ch_o_exp: VAR To_syntax %("ch",o,exp)
2  OutC(ch_o_exp): process = H({pre|pre'ok1=true AND
3    ((pre'wait1=true AND FORALL (v:int):(NOT pre'ref1(input((ch_o_exp'1,i,v)))) AND
4      pre'tr1=pre'tr) OR
5      (pre'wait1=false AND EXISTS(s:S): (pre'tr1=append(pre'tr,
6        cons(s_e(s,output((ch_o_exp'1,ch_o_exp'2,aeval(ch_o_exp'3)(s)))),null))))))});
7  >>(ch_o_exp, P): process = (OutC(ch_o_exp) ++ P);
```

In our proposed denotational semantics, predicate $ch? \notin ref'$ denotes a process refuses all inputs. Here, we constrain explicitly that no input is in the refused set (shown at line 3).

Different from the above encoding of synchronous channel output, the formalisation of synchronous channel input below takes parametric process P_i into account. We model the input prefixing by a set of observational variable records, where each record is a member of the sequential composition of a channel input process $InC(ch_i_m, v)$ and process $P_i(v)$. Value v denotes a possible message. In this way, parametric process P_i can also be applied to multiple indexed processes, for example, process P_i can be of the type $[int, \dots, int \rightarrow process]$.

```

ch_i_m: VAR Ti_syntax %("ch",i,m)
InC(ch_i_m, v): process = H({pre|pre'ok1=true AND
  ((pre'wait1=true AND FORALL (v1:int):(NOT pre'ref1(output((ch_i_m'1,o,v1)))) AND
    pre'tr1=pre'tr) OR
    (pre'wait1=false AND EXISTS(s:S):(pre'tr1=append(pre'tr,
      cons(s_e(s,input((ch_i_m'1,ch_i_m'2,v))),null))))))});
>>(ch_i_m, Pi): process = H({pre|EXISTS(v:int):member(pre,(InC(ch_i_m, v) ++ Pi(v))});

```

5.3.5 Data Operation Prefixing

To formalise the semantics of data operation prefixing in PVS, we first define the syntax for the non-communicating event associated with sequential program. Such data operation is of the type *Tuple* with two elements, [Tnc, Prog], one is an uninterpreted type denoting non-communicating events, and the other defines the syntax for sequential programs.

```

Tnc: TYPE % non-communicating event
Td_syntax: TYPE = [Tnc, Prog]
e_prog: VAR Td_syntax %(e,prog)
Data(e_prog): process = H({pre|pre'ok1=true AND EXISTS(s:S):
  ((pre'wait1=true AND ceval(e_prog'2)(s)=is_abort(bot) AND
    pre'tr1=append(pre'tr,cons(s_t(s,is_abort(bot)),null))) OR
    (pre'wait1=false AND ceval(e_prog'2)(s)/=is_abort(bot) AND
      EXISTS(s1:S):(ceval(e_prog'2)(s)=is_S(s1) AND
        pre'tr1=append(pre'tr,cons(s_t(s,is_S(s1)),null))))));
>>(e_prog, P): process = (Data(e_prog) ++ P);

```

5.3.6 Choice

The internal choice indicates that process $P \sqcap Q$ behaves like either P or Q .

```

\/(P, Q): process = H({pre|P(pre) OR Q(pre)});

```

Regarding the external choice of two processes P and Q , if no observation has been made and termination has not occurred (i.e., the predicate of process *Stop* being true as indicated by **Stop(pre)**), then it behaves like the conjunction of P and Q , or otherwise, it behaves as the disjunction. The formalisation of external choice is shown below.

$$\wedge(P, Q): \text{process} = H(\{\text{pre} \mid (\text{Stop}(\text{pre}) \text{ AND } P(\text{pre}) \text{ AND } Q(\text{pre})) \text{ OR} \\ (\text{NOT } \text{Stop}(\text{pre}) \text{ AND } (P(\text{pre}) \text{ OR } Q(\text{pre})))\});$$

5.3.7 State Guard

The behaviour of process $[b]P$ is determined by the evaluation of boolean condition b . The evaluation is modelled by function **g_beval**, which first checks whether two input traces fulfil an extension relationship (specified by an overloading function $<(\mathbf{t}, \mathbf{t1})$ for trace prefixing). If no, **g_beval** returns false. Otherwise, **g_beval** evaluates b (**beval(b)**) under the pre-state of the first element from the extended trace performed by process P . As a trace (in our denotational semantics) records two kinds of observations, i.e., state transition and event occurrence, its evaluation thus consists of two parts, namely, $\text{length}(\mathbf{t}) < \text{length}(\mathbf{t1}) \text{ AND } \mathbf{s_t?}(\text{nth}((\mathbf{t1}-\mathbf{t}), 0))$ and $\text{length}(\mathbf{t}) < \text{length}(\mathbf{t1}) \text{ AND } \mathbf{s_e?}(\text{nth}((\mathbf{t1}-\mathbf{t}), 0))$, where the PVS function **nth** returns the $(n + 1)$ th element from the trace. For example, $\text{nth}((\mathbf{t1} - \mathbf{t}), 0)$ represents the predicate $\text{head}(t1 - t)$ describing the initial observation on the trace of a process.

```

b: VAR Bexp
t, t1: VAR Trace
g_beval(b, t, t1): bool =
  IF <(t,t1) THEN
    IF length(t) < length(t1) AND s_t?(nth((t1-t),0)) THEN
      beval (b) ((s_s1(nth((t1-t),0)))'1)
    ELSE
      (IF length(t) < length(t1) AND s_e?(nth((t1-t),0)) THEN
        beval (b) ((s_event(nth((t1-t),0)))'1)
      ELSE false
      ENDIF)
    ENDIF
  ELSE false
  ENDIF;
[[|](b,P):process = H({pre|(g_beval(b,pre'tr,pre'tr1)=true AND P(pre)) OR
  (g_beval(b,pre'tr,pre'tr1)=false AND Stop(pre))});

```

5.3.8 Parallel Composition

As mentioned in Chapter 4, Section 4.1.3.8, the semantics of parallel composition captures different types of merge. We model parallel composition in PVS below.

```

1 P, Q: VAR process
2 X: VAR set[RefE]
3 Par(P, Q)(X): process = H({pre|EXISTS (p,q: Obs, X1,X2: set[RefE]): P(p) AND Q(q) AND
4   pre'ok=p'ok AND pre'wait=p'wait AND pre'ref=p'ref AND pre'tr=p'tr AND
5   pre'ok=q'ok AND pre'wait=q'wait AND pre'ref=q'ref AND pre'tr=q'tr AND
6   pre'ok1=(p'ok1 AND q'ok1) AND pre'wait1=(p'wait1 OR q'wait1) AND
7   pre'ref1=union(union(inter(inter(p'ref1,q'ref1),X2),
8     inter(union(p'ref1,q'ref1), X1)), (inter(p'ref1,q'ref1)-X1-X2)) AND
9   X1={x:RefE|action?(x) AND X(x) AND (NOT input?(x)) AND (NOT output?(x))} AND
10  X2={x:RefE|(input?(x) OR output?(x)) AND X(x) AND (NOT action?(x))} AND
11  member(pre'tr1-pre'tr, tr_syn(p'tr1-p'tr, q'tr1-q'tr, X))})

```

In the above formalisation, the value of an undashed observational variable of the parallel composition is the same as the counterpart of processes P and Q , captured by a list of conjunctions at lines 4-5. On the other hand, the dashed variables capture four kinds of behaviour: 1. the divergence (`pre'ok1=(p'ok1 AND q'ok1)` at line 6), 2. the termination (`pre'wait1=(p'wait1 OR q'wait1)` at line 6), 3. the refusal (at lines 7-10) where we derive the set $X1$ and $X2$ from the set X , $X1$ denotes common actions, and $X2$ denotes synchronous channel inputs and outputs, 4. the trace of the composition is a member of the set of traces produced by the trace synchronisation function `tr_syn` at line 11.

We remark that the definition of trace synchronisation function is composed of nine cases from six groups. In PVS, we formalise the definition in terms of all seven trace types: an empty trace, a non-empty trace consisting of state transitions, actions in the set X , actions not in X , synchronous channel outputs/inputs/communications. That is to say, our formalisation captures 49 scenarios of two traces. Considering the page limit, we illustrate the function definition corresponding to the different cases with the code fragment.

Empty Trace The specification below covers the scenarios where the first input is an empty trace; we use `null` to indicate an empty trace. Line 5 models the result when both input traces are empty (**case-1**). When the second input trace is not empty, the result is determined by the first observation of the second input trace. Lines 9-13 divide **case-2** where the observation is an action (checked by recogniser `action?`) into two cases depending on whether the action is in the set X (checked by `X((s_event(se))'2)`). Lines 15-16 depict **case-3** where the observation is a synchronous channel input/output/communication, and lines 6-8 are for **case-4** when the observation is a state transition.

```

1  tr1, tr2: VAR Trace
2  tr_syn(tr1, tr2, X): RECURSIVE set[Trace] =
3    CASES tr1 OF
4      null: CASES tr2 OF
5        null: {t1: Trace|t1=null},
6        cons(se,t): IF s_t?(se) THEN
7          {t1:Trace|EXISTS(l:Trace): t1=cons(se,l) AND member(l,
8            tr_syn(tr1,t,X))}
9          ELSE (IF action?((s_event(se))'2) THEN
10             IF X((s_event(se))'2) THEN
11               {t1: Trace|t1=null}
12             ELSE {t1:Trace|EXISTS(l:Trace): t1=cons(se,l) AND
13               member(l, tr_syn(tr1,t,X))}
14             ENDIF
15             ELSE {t1:Trace|EXISTS(l:Trace): t1=cons(se,l) AND
16               member(l, tr_syn(tr1,t,X))}
17             ENDIF)
18          ENDIF
19        ENDCASES,
20    ...
21  ENDCASES
22  MEASURE length(tr1)+length(tr2)

```

Synchronous Channel Input and Output The following fragment formalises **case-5**, i.e., the synchronisation between one synchronous channel input and output. Line 9 denotes the match condition which checks the equivalence of three entities: the pre-states of two input traces ($s1'1 = s2'1$), the channel names ($c1'1 = (co(s2'2))'1$), and the messages through the channel ($c1'3 = (co(s2'2))'3$). When the match condition is valid, there are two possible behaviours depending on whether a synchronous channel communication occurs (lines 10-14) or not (lines 15-16). Otherwise, lines 17-18 handle the case when the condition is invalid.

```

1  tr_syn(tr1, tr2, X): RECURSIVE set[Trace] =
2    CASES tr1 OF
3      cons(se1,t1): CASES se1 OF
4        s_e(s1): CASES s1'2 OF
5          input(c1): CASES tr2 OF
6            cons(se2,t2): CASES se2 OF
7              s_e(s2):
8                IF output?(s2'2) THEN
9                  (IF s1'1=s2'1 AND c1'1=(co(s2'2))'1 AND c1'3=(co(s2'2))'3 THEN
10                     {t3:Trace|(EXISTS(se3:SE)(s3:StateEvent)(l:Trace):s_e?(se3)
11                       AND (s_event(se3))'1=s1'1 AND chancom?((s_event(se3))'2)
12                       AND (cm((s_event(se3))'2))'1=c1'1 AND (cm((s_event(se3))'2))'3=c1'3
13                       AND (cm((s_event(se3))'2))'2=m
14                       AND t3= cons(se3,1) AND member(1, tr_syn(t1,t2,X)))
15                     OR (EXISTS(l: Trace): (t3=cons(se1,1) AND member(1,tr_syn(t1,tr2,X)))
16                     OR (t3=cons(se2,1) AND member(1, tr_syn(tr1,t2,X))))}
17                   ELSE {t3:Trace|EXISTS(l: Trace): (t3=cons(se1,1) AND member(1,tr_syn(t1,tr2,X)))
18                     OR (t3=cons(se2,1) AND member(1, tr_syn(tr1,t2,X)))}
19                 ENDIF)
20            ELSE ...
21          ...
22        ENDCASES
23  MEASURE length(tr1)+length(tr2)

```

Both Actions When encoding **case-6**, namely, synchronisation between two actions, there are five scenarios: 1. both actions are in the set X but different (lines 9-11), 2. actions are the same but from different pre-states (lines 12), 3. actions are the same and from the same pre-states (lines 13-14), 4. an action is not in X (line 16), and 5. both actions are not in X (lines 25, 27-28).

```

1  tr_syn(tr1, tr2, X): RECURSIVE set[Trace] =
2    CASES tr1 OF
3      cons(se1,t1): CASES se1 OF
4        s_e(s1): CASES s1'2 OF
5          action(a): IF X(action(a)) THEN
6            CASES tr2 OF
7              cons(se2,t2): CASES se2 OF
8                s_e(s2):
9          IF action?(s2'2) THEN
10         IF X(s2'2) THEN
11           IF action(a)=s2'2 THEN {t3:Trace|t3=null}
12           ELSE (IF s1'1/=s2'1 THEN {t3:Trace|t3=null}
13             ELSE {t3:Trace|EXISTS(l:Trace): t3=cons(se1,l) AND
14               member(l, tr_syn(t1,t2,X))}) ENDIF)
15         ENDIF
16       ELSE {t3:Trace|EXISTS(l:Trace): t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X))}
17       ENDIF
18     ELSE ...
19     ...
20           ELSE % action(a) is not in the set X
21           CASES tr2 OF
22             cons(se2,t2): CASES se2 OF
23               s_e(s2):
24             IF action?(s2'2) THEN
25               IF X(s2'2) THEN ...
26             ELSE {t3:Trace|EXISTS(l: Trace): (t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X)))
27               OR (t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X)))}
28             ...
29           ENDCASES
30   MEASURE length(tr1)+length(tr2)

```

Action and Other Type When a synchronisation is between one action and an event or a state transition, the result is determined by the action. If it is in the set X , then its

occurrence is postponed (line 10), or otherwise, either observation occurs (lines 17-18).

```

1  tr_syn(tr1, tr2, X): RECURSIVE set[Trace] =
2    CASES tr1 OF
3      cons(se1,t1): CASES se1 OF
4        s_e(s1): CASES s1'2 OF
5          action(a): IF X(action(a)) THEN
6            CASES tr2 OF
7              cons(se2,t2): CASES se2 OF
8                s_e(s2):
9          IF action?(s2'2) THEN ...
10         ELSE {t3:Trace|EXISTS(l:Trace): t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X))}
11         ...
12         ELSE % action(a) is not in the set X
13         CASES tr2 OF
14           cons(se2,t2): CASES se2 OF
15             s_e(s2):
16         IF action?(s2'2) THEN ...
17         ELSE {t3:Trace|EXISTS(l: Trace): (t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X)))
18           OR (t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X)))}
19         ...
20     ENDCASES
21  MEASURE length(tr1)+length(tr2)

```

State Transition and Other Type When a synchronisation is between two state transitions (lines 6-7), or between a state transition and a channel input/output/communication (lines 9-11), either observation can occur.

```

1 tr_syn(tr1, tr2, X): RECURSIVE set[Trace] =
2   CASES tr1 OF
3     cons(se1,t1): CASES se1 OF
4       s_t(s1): CASES tr2 OF
5         cons(se2,t2): CASES se2 OF
6         s_t(s2):{t3:Trace|EXISTS(l: Trace): (t3=cons(se1,l) AND member(l,tr_syn(t1,tr2,X)))
7           OR (t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X)))},
8         s_e(s2): IF action?(s2'2) THEN ...
9           ELSE {t3:Trace|EXISTS(l: Trace): (t3=cons(se1,l) AND
10             member(l,tr_syn(t1,tr2,X)))
11             OR (t3=cons(se2,l) AND member(l, tr_syn(tr1,t2,X)))}
12         ...
13   ENDCASES
14 MEASURE length(tr1)+length(tr2)

```

5.3.9 Interleave

The semantics of interleaving process $P \parallel Q$ is similar to the semantics of parallel operator, except the set X which only contains synchronous channel outputs and inputs.

```
Inter(P, Q)(X): process = Par(P, Q)(X)
```

5.3.10 Hiding

In process $P \setminus X_1$, all the occurrence of actions in set X_1 are not observed or controlled by the environment. We formalise the hiding operator as follows.

```

1 X1: VAR set[Ta]
2 Hid(P,X1): process = H({pre|EXISTS (s:Obs, X:set[RefE]): P(s) AND
3   pre'ok=s'ok AND pre'wait=s'wait AND pre'ref=s'ref AND pre'tr=s'tr AND pre'ok1=s'ok1
4   AND pre'wait1=s'wait1 AND pre'tr1=append(pre'tr, hide(s'tr1-s'tr,X1)) AND
5   pre'ref1=s'ref1-X AND X={x:RefE|action?(x) AND X1(ac(x)) AND (NOT input?(x)) AND
6   (NOT output?(x))})})

```

The above definition depicts that the behaviour of hiding process is the same as P except the final value of refusals and traces. Specifically, the refusals are defined as that from P excluding the hiding actions in X . Since the refusals has the type `set[RefE]` and set $X1$ is of the type `set[Ta]`, we cannot directly use the set extraction function in PVS. Instead, we construct a set of elements of the type `RefE` containing only actions (specified by lines 5-6). To model the actions hiding in a trace, we define a function `hide` which is a recursive function over the trace structure.

```

hide(t,X1): RECURSIVE Trace =
  CASES t OF
    null: null,
    cons(x1, x2): (IF s_e?(x1) THEN
      IF action?(s_event(x1)'2) THEN
        IF X1(ac((s_event(x1))'2)) THEN
          hide(x2,X1)
        ELSE cons(x1,hide(x2,X1))
        ENDIF
      ELSE cons(x1,hide(x2,X1))
      ENDIF
    ELSE cons(x1,hide(x2,X1))
    ENDIF)
  ENDCASES
  MEASURE length(t)

```

Here, we use PVS `CASES` expressions to discuss two patterns of the trace. If the trace contains an action from set $X1$ (specified as `X1(ac((s_event(x1))'2))`), we remove this action from the trace (specified as `hide(x2,X1)`). Otherwise, this trace is unchanged (specified as `cons(x1,hide(x2,X1))`).

5.3.11 Refinement

We define symbol $|>$ to represent the refinement operator (\sqsubseteq) in the following PVS specifications, where the PVS predefined function `subset?` checks whether set P is a subset of set Q . Namely, process P refines Q iff the formalisation P is a subset of Q in our encoding.

```
|>(P,Q): bool = subset?(P,Q)
```

5.3.12 Recursion

In $\text{CSP}\#$, the semantics of recursion process is defined using the weakest fixed point, which is the greatest lower bound of all the fixed points. We define the fixed point theory below where $\text{glb}(\text{SX})$ denotes the greatest lower bound of any set SX of processes, $\text{monotonic?}(F)$ checks whether F is a monotonic mapping, and $\text{mu}(G)$ represents the weakest fixed point given a monotonic mapping G .

```
SX: VAR set[process]
X, Y: VAR process
pre: VAR Obs
glb(SX): process = H({pre|FORALL (X: (SX)): X(pre)})
F: VAR [process -> process]
monotonic?(F): bool = FORALL X,Y: X |> Y IMPLIES F(X) |> F(Y)
G: VAR (monotonic?)
mu(G): process = glb({X| X |> G(X)})
```

The main properties of glb and mu are provided below and they have been proved in PVS.

```

glb_is_bound_L1A: LEMMA FORALL (X: (SX)): X |> glb(SX)

glb_is_sup_L1B: LEMMA (FORALL (X: (SX)): X |> Y) IMPLIES glb(SX) |> Y

closure_mu: LEMMA mu(G) |> G(mu(G))

smallest_closed: LEMMA X|> G(X) IMPLIES X |> mu(G)

fixed_point: LEMMA G(mu(G)) = mu(G)

weakest_fixed_point: LEMMA G(X) = X IMPLIES X |> mu(G)

```

5.4 Mechanical Proof of Laws

So far, we have formalised our denotational semantic model and process semantics in PVS. In this section, we apply the PVS typechecker to validating the consistency of the denotational semantics and the PVS prover to mechanically proving essential laws of our formalisations so as to check the correctness of our encoding.

Typechecker in PVS analyses the theory for semantic consistency [55]. It usually checks the semantic constraints, determines the types of expressions, and resolves names. After type-checking, proof obligations, called type correctness conditions(TCCs) are generated which are mostly related to predicate subtypes and termination in the recursive definitions. In our work, we have discharged 84 TCCs for the process semantics theory. These TCCs are mainly from the subtypes requiring that every $\text{CSP}\#$ process satisfies the idempotent property and terminations of recursive definitions such as trace synchronisation function.

Based on our semantic formalisation in PVS, we can derive many important properties. We have mechanically proved a set of important laws that are essential in the verification of $\text{CSP}\#$ programs. Regarding the properties of healthiness conditions, we have proved that conditions **R1**, **R3**, **CSP1**, and **CSP2** are idempotent and commutative. For example, the following PVS proof script is used to prove the commutativity property of **R1** and **CSP1**.

```
R1_CSP1_commutative :
```

```
|-----
```

```
1  FORALL (P: Relation): CSP1(R1(P)) = R1(CSP1(P))
```

```
Rule? (skolem!)
```

```
Skolemizing,
```

```
this simplifies to:
```

```
R1_CSP1_commutative :
```

```
|-----
```

```
1  CSP1(R1(P!1)) = R1(CSP1(P!1))
```

```
Rule? (expand* "CSP1" "R1")
```

```
Expanding the definition(s) of (CSP1 R1),
```

```
this simplifies to:
```

```
R1_CSP1_commutative :
```

```
|-----
```

```
1  (pre: Obs |
```

```
    (NOT pre'ok AND pre'tr <= pre'tr1) OR P!1(pre) AND pre'tr <= pre'tr1)
```

```
=
```

```
(pre_1: Obs |
```

```
    ((NOT pre_1'ok AND pre_1'tr <= pre_1'tr1) OR P!1(pre_1))
```

```
    AND pre_1'tr <= pre_1'tr1)
```

```
Rule? (apply-extensionality :hide? t)
```

```
Applying extensionality,
```

```
this simplifies to:
```

```

R1_CSP1_commutative :

  |-----
1  ((NOT x!1'ok AND x!1'tr <= x!1'tr1) OR P!1(x!1) AND x!1'tr <= x!1'tr1)
   =
   ((NOT x!1'ok AND x!1'tr <= x!1'tr1) OR P!1(x!1)) AND x!1'tr <= x!1'tr1)

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.

```

In the above proof script, command `skolem!` introduces Skolem constant for the universally quantified variable `P` in the lemma, command `expand*` expands the definitions of `CSP1` and `R1`, command `apply-extensionality :hide? t` uses extensionality to prove equality, and command `grind` installs rewrites and repeatedly applies simplification.

From these lemmas, we show that our formalisation of condition `H` is idempotent and every `CSP#` process is healthy. We have also proved that some important algebraic laws of `CSP#` processes: internal choice is commutative and idempotent; the parallel composition is commutative (`par-1`); etc.

5.5 Summary

In this chapter, we encoded the denotational semantics of `CSP#` into the PVS theorem prover. The type consistency of mechanised semantics was validated by proving the TCCs generated from typechecking. The mechanisation provides a framework for developing mechanical verification for `CSP#` specifications, for example, to check process equivalence based on the formalised refinement relation. Based on the encoding, we also proved properties of healthiness conditions and algebraic laws related to process definitions in PVS, which can act as auxiliary reasoning rules to improve verification automation. Our framework is conve-

nient for specifying and analysing complex systems, and it can be easily extended to analyse similar concurrency languages.

The work in this chapter is related to the mechanisation of various CSP models. Camilleri [11] encoded the trace model of CSP and later a variation of the failures-divergences model [12] into the HOL system [25]. Dutertre and Schneider [21] formalised the trace model of CSP in PVS, tailored to reason about security protocols; Wei and Heather [91] extended this formalisation to the stable-failures model in order to verify liveness properties. Tej and Wolff [88] encoded the failures-divergences model in Isabelle/HOL. Isobe and Roggenbach [34, 35] improved this work with tool support from CSP-Prover which handles more CSP models including trace model, stable-failures model and stable-revivals model [71]. However, all the above formalisation of various CSP models lacks the support of complex data.

There exists other research on encoding denotational semantics of integrated languages with CSP. For example, Oliveira *et al.* [52] presented the mechanisation of the UTP semantics of Circus in ProofPower-Z [60]; the formalised semantics is defined using a set-based theory. Moreover, the mechanical proof of various refinement laws was conducted in [53]. Wei *et al.* [92] encoded the UTP semantics of Timed Circus in PVS, where the formalisation of time operators Delay, Timeout and Deadline was presented. Our work follows a similar way of their encoding, but covers the formalisation of sequential programs such as assignments, sequential composition on shared variables, while Timed Circus supports assignments on local variables only. Moreover, we formalise different event types covering both event synchronisation and pairwise handshake through synchronous channels.

This work is also related to research on formalisation of UTP theories. Feliachi *et al.* [22] formalised a part of UTP theory in Isabelle/HOL including theories of alphabetised relations and designs. Recently Foster and Woodcock [24] improved the mechanisation of UTP theory, by defining a unified type for predicates and supporting more operators and meta-theoretic

proofs; their Isabelle/UTP currently supports theories of relations and designs, and provides a platform for mechanising the semantics for specification languages like CSP.

Chapter 6

Conclusion

In this chapter, we summarize the contributions of this thesis and discuss possible future directions of our work.

6.1 Contributions

This thesis presents systematic research work centred on a highly expressive formal modelling language $\text{CSP}\#$ which supports concurrent systems with global shared variables. The research work includes the conduction of a comprehensive comparison of $\text{CSP}\#$ with other CSP extensions, construction of $\text{CSP}\#$ denotational semantics, and encoding $\text{CSP}\#$ denotational semantics into a theorem prover. Specifically, four main contributions of this thesis are summarized below.

- In Chapter 3, we have compared $\text{CSP}\#$ and CSP_M , popular CSP extensions for concurrent systems, from a wide range of aspects to identify subtle differences. First, we explored the modelling features in terms of language syntax and operational semantics. Second, we investigated the reasoning power of their analysis tools, namely, PAT

for CSP#, FDR and ProB for CSP_M from the view of tool capability and efficiency. The qualitative analysis was made by conducting experiments on eight benchmark systems to verify different properties; the designed benchmarks can also be applied to experiments on the comparison of PAT with other tools. Further we discussed the guideline for choosing an appropriate modelling language and reasoning tools for particular concurrent systems.

- In Chapter 4, we have proposed a denotational semantic model to deal with communication based concurrency and shared-variable based concurrency. We have defined *hybrid* traces (capturing both states and events) for recording process behaviours involving shared variables. Moreover, our model can be adapted/enhanced to define the denotational semantics for other languages which possess similar concurrency mechanisms. Based on this semantic model, we have defined a denotational semantics for CSP# processes. Our definition covers communications through event-based synchronisation and pairwise synchronised handshake over channels. The refinement relationship between processes has also been presented.
- To make the semantics of CSP# more complete, on one hand, we have defined a set of algebraic laws which concern the distinct features of CSP# like shared variables (Chapter 4, Section 4.2). These laws are established based on the denotational semantics, which partially prove the correctness of denotational semantics. On the other hand, we have derived a *closed* semantics from the *open* denotational semantics by focusing on certain particular types of hybrid traces to consider a closed environment (Chapter 4, Section 4.3).
- In Chapter 5, we have mechanised CSP# denotational semantics in the PVS theorem prover, which forms the foundation towards the theorem proving approach to verify complex CSP# models. First, we have encoded the semantic model including observational variables and healthiness conditions. Based on this encoding, we have

formalised the semantics of expressions and sequential programs. Further, we have mechanised the process semantics and refinement relationship. Last but not least, we have applied the PVS typechecker to validating the consistency of the semantics and the PVS prover to mechanically proving essential laws of our formalisations so as to validate the correctness of our encoding.

6.2 Future Work

In this section, we outline some possible extensions to our work presented in this thesis.

We have defined a closed semantics for $\text{CSP}\#$ which considers the situation where the model of the whole system has been built or the behaviour of the environment has been modelled. To ensure the consistency between different semantics of the same language, i.e., $\text{CSP}\#$, our next step is to link the proposed closed denotational semantics of $\text{CSP}\#$ to its operational semantics [83] by applying the step relation in UTP [31].

Regarding the development of a theorem proving framework for $\text{CSP}\#$, we have mechanised the $\text{CSP}\#$ denotational semantics into PVS. An immediate step is to validate more complex laws and properties of the mechanised theories, for example, to cover all algebraic laws in Chapter 4, Section 4.2. Once validated, these laws and properties can serve as auxiliary reasoning rules to improve verification automation in PVS. Moreover, as the ultimate goal of our mechanisation framework is to support verification of concurrent systems modelled in $\text{CSP}\#$, we plan to specify and verify common benchmark system (e.g., the dining philosophers problem); we are keen to leverage induction techniques to deal with infinite state systems.

In our systematic comparison of $\text{CSP}\#$ and another popular CSP extension CSP_M , we have explored the classical model checking techniques used by their respective tools, namely, PAT

for $\text{CSP}\#$, FDR and ProB for CSP_M . In the future, we will extend the comparison to more advanced techniques such as SAT-based FDR [56] and BDD-based PAT [42]. In addition, we have discussed possible translation between $\text{CSP}\#$ and CSP_M models from the comparison. Thus, proving the semantics equivalence of the translation and its implementation are our goals as well.

Beyond $\text{CSP}\#$ which is the focus of this thesis, we can extend our work to handle other models supporting more features such as time and probability. For example, Stateful Timed CSP [86, 84] extends $\text{CSP}\#$ with time process constructs like *timeout* and *deadline*; we can compare Stateful Timed CSP and Timed CSP [68, 73, 20, 74] (which extends Hoare's CSP with real-time constructs like *WAIT*) including their tool support, followed by defining denotational semantics for Stateful Timed CSP.

Bibliography

- [1] <http://www.cs.ox.ac.uk/ucs/CSPtools.html>. 3.4
- [2] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996. 1.1, 3.3
- [3] C. M. Angelo, L. J. M. Claesen, and H. D. Man. Degrees of Formality in Shallow Embedding Hardware Description Languages in HOL. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 89–100. Springer-Verlag, 1994. 5
- [4] C. Attiogbé. Mechanization of an Integrated Approach: Shallow Embedding into SAL/PVS. In *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM 2002)*, pages 120–131, 2002. 5
- [5] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. 5
- [6] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, H. Herbelin, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.1*. INRIA-Rocquencourt-CNRS-ENS Lyon, Dec. 1996. 5
- [7] G. Barrett. Model Checking in Practice: The T9000 Virtual Channel Processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, 1995. 3
- [8] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with Embedding Hardware Description Languages in HOL. In *Proceedings of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, 1992. 5
- [9] P. J. Broadfoot and A. W. Roscoe. Tutorial on FDR and Its Applications. In *SPIN Model Checking and Software Verification*, page 322. Springer Berlin Heidelberg, 2000. 1.1
- [10] S. D. Brookes. Full Abstraction for a Shared-Variable Parallel Language. *Information and Computation*, 127(2):145–163, 1996. 4.4

- [11] A. J. Camilleri. Mechanizing CSP Trace Theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990. 5.5
- [12] A. J. Camilleri. A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics. In *IV Higher Order Workshop, Banff 1990*, pages 123–150. Springer London, 1991. 5.5
- [13] G. H. P. Carvalho, T. Dias, A. Mota, and A. Sampaio. Analytical Comparison of Refinement Checkers. In *SBMF*, pages 61–66, 2011. 3.4
- [14] A. Cavalcanti and J. Woodcock. A Tutorial Introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006. 4.1.1.2, 4.4
- [15] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., 1997. 5
- [16] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000. 5
- [17] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996. 1.1
- [18] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009. 1.1
- [19] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, 1995. 2.3
- [20] J. Davies. *Specification and Proof in Real-Time Systems*. PhD thesis, Oxford University, 1991. 6.2
- [21] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *TPHOLs*, pages 121–136. Springer, 1997. 5.5
- [22] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying Theories in Isabelle/HOL. In *Unifying Theories of Programming*, volume 6445 of *Lecture Notes in Computer Science*, pages 188–206. Springer Berlin Heidelberg, 2010. 5.5
- [23] C. Fischer. Combining Object-Z and CSP. In *FBT*, pages 119–128, 1997. 1.1
- [24] S. Foster and J. Woodcock. Unifying Theories of Programming in Isabelle. In *Unifying Theories of Programming and Formal Engineering Methods*, volume 8050 of *Lecture Notes in Computer Science*, pages 109–155. Springer Berlin Heidelberg, 2013. 5.5

- [25] M. J. C. Gordon. HOL: a proof generating system for Higher Order Logic. In *VLSI Specification, Verification and Synthesis*, pages 73–128. 1988. 5.5
- [26] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993. 5
- [27] A. Hall and R. Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, 19:18–25, 2002. 3
- [28] J. Helin. Combining Deep and Shallow Embeddings. *Electronic Notes in Theoretical Computer Science*, 164(2):61–79, 2006. 5
- [29] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 1.1, 2.1, 3.1.2, 4.4
- [30] C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. Sufrin. Laws of Programming. *Communications of the ACM*, 30(8):672–686, 1987. 1.1
- [31] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998. 1.1, 2.2, 4, 4.1.1.1, 4.1.1.2, 4.1.3.8, 4.4, 5.3, 6.2
- [32] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003. 3.1.2
- [33] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), 1992. 2.2
- [34] Y. Isobe and M. Roggenbach. A Generic Theorem Prover of CSP Refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 108–123. Springer Berlin Heidelberg, 2005. 5.5
- [35] Y. Isobe and M. Roggenbach. Proof Principles of CSP – CSP-Prover in Practice. In *Dynamics in Logistics*, pages 425–442. Springer Berlin Heidelberg, 2008. 5.5
- [36] C. B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., 1990. 1.1
- [37] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, volume 2805, pages 855–874. Springer Berlin Heidelberg, 2003. 1.1, 3
- [38] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 278–297. Springer-Verlag, 2008. 3.1.1, 3.3

- [39] M. Leuschel, T. Massart, and A. Currie. How to make FDR Spin LTL model checking of CSP by refinement. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 99–118. Springer Berlin Heidelberg, 2001. 3.3.1
- [40] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model Checking Linearizability via Refinement. In *Proceedings of the 2Nd World Congress on Formal Methods, FM '09*, pages 321–337. Springer-Verlag, 2009. 3.3
- [41] Y. Liu, W. Chen, Y. A. Liu, J. Sun, S. J. Zhang, and J. S. Dong. Verifying Linearizability via Optimized Refinement Checking. *IEEE Transactions on Software Engineering*, 39(7):1018–1039, 2013. 3.3, 3.3.2, A.7
- [42] Y. Liu, J. Sun, and J. S. Dong. PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers. In *IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE 2011)*, pages 190–199, 2011. 6.2
- [43] G. Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Formal Aspects of Computing*, 20(3):277–294, 2008. 3.3.1, 3.3.2
- [44] G. Lowe and B. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23:659–669, 1997. 1.1
- [45] F. S. E. Ltd. *Failures-Divergence Refinement - FDR2 User Manual (version 2.91)*, 2010. 1.1, 3, 3.1.2, 3.3.1
- [46] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000. 1.1
- [47] J. McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962. 2.2
- [48] A. A. McEwan and J. Woodcock. Unifying Theories of Interrupts. In *Proceedings of the 2nd International Conference on Unifying Theories of Programming, UTP'08*, pages 122–141. Springer-Verlag, 2010. 4.1.3
- [49] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989. 1.1, 3.3.2, A.3
- [50] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990. 2.2
- [51] T. Murray. On the limits of refinement-testing for model-checking CSP. *Formal Aspects of Computing*, 25(2), 2013. 3.3.1
- [52] M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying Theories in Proofpower-Z. In *Proceedings of the First International Conference on Unifying Theories of Programming, UTP'06*, pages 123–140. Springer-Verlag, 2006. 5.5
- [53] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3–32, 2009. 4.4, 5.5

- [54] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence, pages 748–752. Springer-Verlag, 1992. 1.1, 2.3, 5
- [55] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. SRI International, November 2001. 5.4
- [56] H. Palikareva, J. Ouaknine, and A. W. Roscoe. Faster FDR Counterexample Generation Using SAT-Solving. *Electronic Communications of the EASST*, 23, 2009. 3.3.2, 6.2
- [57] A. N. Parashkevov and J. Yantchev. ARC - A tool for efficient refinement and equivalence checking for CSP. In *IEEE International Conference on Algorithms and Architectures for Parallel Processing ICA3PP '96*, pages 68–75, 1996. 3.4
- [58] L. C. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994. 5
- [59] J. L. Peterson and A. Silberschatz. *Operating System Concepts (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1985. 3.3.2, A.8
- [60] ProofPower. <http://www.lemma-one.com/proofpower/index/index.html>. 5.5
- [61] S. Qin, J. S. Dong, and W.-N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 321–340. Springer Berlin Heidelberg, 2003. 4.4
- [62] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997. 1.1, 2.1.2, 3.2, 3.2.11, 3.3.1, 4.4
- [63] A. W. Roscoe. On the expressive power of CSP refinement. *Formal Aspects of Computing*, 17:93–112, 2005. 3.3.1
- [64] A. W. Roscoe. Revivals, stuckness and the hierarchy of CSP models. *Journal of Logic and Algebraic Programming*, 78(3):163–190, 2009. 3.3.1
- [65] A. W. Roscoe. CSP is Expressive Enough for Pi. In *Reflections on the work of C.A.R. Hoare*. Springer, 2010. 3.1.2
- [66] A. W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., 2010. 3.1.2, 3.2.4, 3.3.1, 3.3.2
- [67] A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS '95*, pages 133–152, 1995. 3.3.2

- [68] A. W. Roscoe and G. M. Reed. A Timed Model for Communicating Sequential Processes. *Theoretical Computer Science*, 58:249–261, 1988. 6.2
- [69] J. Rushby. Theorem proving for verification. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 39–57. Springer Berlin Heidelberg, 2001. 5
- [70] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and A. W. Roscoe. *Modelling and Analysis of Security Protocols*. Addison-Wesley Professional, 2000. 1.1
- [71] D. G. Samuel, M. Roggenbach, and Y. Isobe. The Stable Revivals Model in CSP-Prover. *Electronic Notes in Theoretical Computer Science*, 250(2):119–134, 2009. 5.5
- [72] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, University of Oxford, 1998. 3.2
- [73] S. Schneider. *Correctness and Communication in Real-Time Systems*. PhD thesis, Oxford University, 1989. 6.2
- [74] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley and Sons, 2000. 1.1, 6.2
- [75] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. SRI International, November 2001. 2.3
- [76] L. Shi and Y. Liu. Modeling and Verification of Transmission Protocols: A Case Study on CSMA/CD Protocol. In *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, SSIRI-C'10, pages 143–149. IEEE Computer Society, 2010. 1.3
- [77] L. Shi, Y. Liu, J. Sun, J. S. Dong, and G. Carvalho. An Analytical and Experimental Comparison of CSP Extensions and Tools. In *Proceedings of the 14th International Conference on Formal Engineering Methods (ICFEM'12)*, volume 7635 of *Lecture Notes in Computer Science*, pages 381–397. Springer Berlin Heidelberg, 2012. 1.3
- [78] L. Shi, Y. Zhao, Y. Liu, J. Sun, J. S. Dong, and S. Qin. A UTP Semantics for Communicating Processes with Shared Variables. In *Proceedings of the 15th International Conference on Formal Engineering Methods (ICFEM'13)*, volume 8144 of *Lecture Notes in Computer Science*, pages 215–230. Springer Berlin Heidelberg, 2013. 1.3
- [79] G. Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313, pages 62–81. Springer Berlin Heidelberg, 1997. 1.1
- [80] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000. 1.1

- [81] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988. 1.1
- [82] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008. 1.1, 3.3
- [83] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating Specification and Programs for System Modeling and Verification. In *The 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, pages 127–135. IEEE Computer Society, 2009. 1.1, 2.1, 3.1.2, 3.2, 4, 6.2
- [84] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and É. André. Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):3:1–3:29, 2013. 1.3, 6.2
- [85] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *The 21st International Conference on Computer Aided Verification*, volume 5643 of *CAV'09*, pages 709–714. Springer, 2009. 1.1, 2.1, 3, 3.3.1
- [86] J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 581–600. Springer, 2009. 6.2
- [87] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair Model Checking with Process Counter Abstraction. In *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 123–139. Springer Berlin Heidelberg, 2009. 3.3.1
- [88] H. Tej and B. Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337. Springer Berlin Heidelberg, 1997. 5.5
- [89] J. D. Ullman. *Elements of ML programming (ML97 ed.)*. Prentice-Hall, Inc., 1998. 2.2
- [90] T. Wang, S. Song, J. Sun, Y. Liu, J. S. Dong, X. Wang, and S. Li. More anti-chain based refinement checking. In *Proceedings of the 14th International Conference on Formal Engineering Methods (ICFEM'12)*, volume 7635 of *Lecture Notes in Computer Science*, pages 364–380. Springer Berlin Heidelberg, 2012. 3.3.1
- [91] K. Wei and J. Heather. Embedding the stable failures model of CSP in PVS. In *Proceedings of the 5th International Conference on Integrated Formal Methods, IFM'05*, pages 246–265. Springer-Verlag, 2005. 5.5

- [92] K. Wei, J. Woodcock, and A. Burns. Embedding the Timed Circus in PVS. Technical report, University of York, 2009. 5.5
- [93] M. Wildmoser and T. Nipkow. Certifying Machine Code Safety: Shallow versus Deep Embedding. In *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, pages 305–320. Springer, 2004. 5
- [94] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. 1.1, 4.1.3.4
- [95] J. Woodcock and A. Cavalcanti. A Concurrent Language for Refinement. In *Proceedings of the 5th Irish Conference on Formal Methods, IW-FM’01*, pages 93–115. British Computer Society, 2001. 1.1
- [96] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4), 2009. 1.1, 3
- [97] H. Zhu, J. P. Bowen, and J. He. From Operational Semantics to Denotational Semantics for Verilog. In *CHARME*, pages 449–466, 2001. 4.4
- [98] H. Zhu, J. He, and J. P. Bowen. From algebraic semantics to denotational semantics for Verilog. *Innovations in Systems and Software Engineering*, 4(4):341–360, 2008. 4.4
- [99] H. Zhu, S. Qin, J. He, and J. P. Bowen. PTSC: probability, time and shared-variable concurrency. *Innovations in Systems and Software Engineering*, 5(4):271–284, 2009. 4.4
- [100] H. Zhu, F. Yang, J. He, J. P. Bowen, J. W. Sanders, and S. Qin. Linking Operational Semantics and Algebraic Semantics for a Probabilistic Timed Shared-Variable Language. *The Journal of Logic and Algebraic Programming*, 81(1):2–25, 2012. 4.4

Appendix A

CSP# Models for Benchmark Systems

This appendix contains the CSP# models of the benchmark systems used in Chapter 3, Section 3.3.2. Each subsection below describes one benchmark system, specifically, an overview of the benchmark system and desired properties, followed by the corresponding CSP# model.

A.1 Readers-writers Problem

Readers-writers problem deals with the situations where many threads may access the same shared memory at a time for reading and writing. A natural constraint is that no process can access the shared memory for reading or writing when another process is writing to the same memory. In this model, a controller is used to guarantee the correct coordination among multiple readers/writers. The verified properties are trace refinement and a LTL property.

```

//////////////////The Model////////////////////////////////
#define M 200;
Writer()    = startwrite → stopwrite → Writer();
Reader()    = startread → stopread → Reader();
Reading(i)  = [i == 0]Controller() □
              [i == M] stopread → Reading(i - 1) □
              [i > 0 && i < M] (startread → Reading(i + 1) □
                               stopread → Reading(i - 1));
Controller() = startread → Reading(1)
               □ stopread → error → Controller()
               □ startwrite → (stopwrite → Controller() □
                               stopread → error → Controller());
ReadersWriters() = Controller() || (|| {M} @ (Reader() || Writer()));
Implementation() = ReadersWriters() \ {startread, stopread, startwrite, stopwrite};
Specification() = error → Specification();
#alphabet Reading {startread, stopread};
//////////////////The Properties////////////////////////////////
#assert Implementation() refines Specification();
#assert ReadersWriters() ⊨ □!error;

```

A.2 Dinning Philosopher

In our model, five philosophers sit around a table. Each philosopher can eat the meal if and only if he/she picks forks on his/her right and left sides, and both folks will be released after the philosopher finishes. This model is deadlock free and the properties are failure refinement and a LTL property.

```

//////////////////The Model////////////////////////////////
#define N 5;
Phil(i) = get.i.i → get.i.(i + 1)%N → eat.i → put.i.(i + 1)%N → put.i.i → Phil(i);
Fork(x) = get.x.x → put.x.x → Fork(x) □
          get.(x + N - 1)%N.x → put.(x + N - 1)%N.x → Fork(x);
Phil0 = get.0.1 → get.0.0 → eat.0 → put.0.1 → put.0.0 → Phil0;
College() = Phil0 || Fork(0) || (|| x : {1..N - 1} @ (Phil(x) || Fork(x)));
Implementation() = College() \ {get.0.0, get.0.1, put.0.0, put.0.1, get.1.1, get.1.2,
                               put.1.1, put.1.2, get.2.2, get.2.3, put.2.2, put.2.3, get.3.3, get.3.4,
                               put.3.3, put.3.4, get.4.4, get.4.0, put.4.4, put.4.0};
Specification() = □ i : {0..N - 1} @ eat.i → Specification();

```

```

//////////////////The Properties//////////////////
#assert Implementation refines < F > Specification;
#assert College()  $\models$   $\Box \Diamond$  eat.0;

```

A.3 Milner's Cyclic Scheduler

This scheduling algorithm is described by Milner in 1989 [49]. N processes are activated in a cyclic manner: process i activates process $i + 1$ for $i < N - 1$ and process $N - 1$ activates process 0. Moreover, a process cannot be re-activated before it has terminated. The property is failures/divergences refinement describing that the implemented scheduler follows a cyclic way.

```

//////////////////The Model//////////////////
#define N 20;
#alphabet Cell {c.i, c.((i+1)%N)};
Cell(i) = ifa (i > 0) {
    c.i  $\rightarrow$  a.i  $\rightarrow$  c.((i+1)%N)  $\rightarrow$  atomic{b.i  $\rightarrow$  Skip}; Cell(i)
} else {
    a.0  $\rightarrow$  c.1  $\rightarrow$  atomic{b.0  $\rightarrow$  Skip}; c.0  $\rightarrow$  Cell(i)
};
MilnerAcyclic() =  $\parallel$  x : {0..N-1} @ Cell(x);
Implementation() = MilnerAcyclic() \ {c.0, b.0, c.1, b.1, c.2, b.2, c.3, b.3, c.4, b.4,
    c.5, b.5, c.6, b.6, c.7, b.7, c.8, b.8, c.9, b.9, c.10, b.10, c.11, b.11, c.12, b.12, c.13, b.13,
    c.14, b.14, c.15, b.15, c.16, b.16, c.17, b.17, c.18, b.18, c.19, b.19};
Spec(n) = a.n  $\rightarrow$  Spec((n+1)%N);
SPECIFICATION() = Spec(0);
//////////////////The Properties//////////////////
#assert Implementation() refines < FD > SPECIFICATION();

```

A.4 The Peg Solitaire Game

The peg solitaire game is a game for one player to move pegs on a board with holes. A valid move is to jump a peg orthogonally over an adjacent peg into a hole which is two positions

away and then remove the jumped peg. There are four orthogonal directions, namely, up, down, left, and right. The goal is to empty the entire board except for a solitary peg in the initial empty hole. A solution is deduced by checking a reachability assertion, where the goal condition specifies that only one peg is on the board at the initial empty hole.

```

////////////////////The Model////////////////////////////////////
#define X - 1;
#define P 1;
#define E 2;
#define initEmptyX 3;
#define initEmptyY 3;
#define W 7;
#define H 7;
var board[H][W] =
    [X, X, P, P, P, X, X,
     X, X, P, P, P, X, X,
     P, P, P, P, P, P, P,
     P, P, P, E, P, P, P,
     P, P, P, P, P, P, P,
     X, X, P, P, P, X, X,
     X, X, P, P, P, X, X];
var pegsCounter = 32;
Up(i, j) = [i - 2 >= 0]([board[i - 2][j] == E &&
    board[i - 1][j] == P]up{board[i - 2][j] = P;
    board[i - 1][j] = E; board[i][j] = E;
    pegsCounter --; } → Game());
Left(i, j) = [j - 2 >= 0]([board[i][j - 2] == E &&
    board[i][j - 1] == P]left{board[i][j - 2] = P;
    board[i][j - 1] = E; board[i][j] = E;
    pegsCounter --; } → Game());
Down(i, j) = [i + 2 < H]([board[i + 2][j] == E &&
    board[i + 1][j] == P]down{board[i + 2][j] = P;
    board[i + 1][j] = E; board[i][j] = E;
    pegsCounter --; } → Game());
Right(i, j) = [j + 2 < W]([board[i][j + 2] == E &&
    board[i][j + 1] == P]right{board[i][j + 2] = P;
    board[i][j + 1] = E; board[i][j] = E;
    pegsCounter --; } → Game());
Peg(i, j) = [board[i][j] == P](Up(i, j) □ Left(i, j) □ Down(i, j) □ Right(i, j));
Peg(i, j) = [board[i][j] == P](Up(i, j) □ Left(i, j) □ Down(i, j) □ Right(i, j));
Game() = □ i : {0..H - 1}@□ j : {0..W - 1}@ Peg(i, j);

```

```

//////////////////The Properties//////////////////
#define goal pegsCounter == 1 && board[initEmptyX][initEmptyY] == P;
#define assert Game() reaches goal;

```

A.5 Knight's Tour

The Knight's tour is a mathematical problem involving a knight on a chessboard. The knight is placed on the empty board, and moving according to the rules of chess, must visit each square exactly once. A knight's tour is called a closed tour if the knight ends on a square attacking the square from which it began (so that it may tour the board again immediately with the same path). Otherwise the tour is open.

```

//////////////////The Model//////////////////
#define N 5;
//the board is a N * N matrix
var board[N * N];
var steps = 0;
//there are 8 ways of jumping
Knight(i, j) = [i - 2 >= 0 && j - 1 >= 0] Move0(i, j) □
               [i - 2 >= 0 && j + 1 < N] Move1(i, j) □
               [i - 1 >= 0 && j - 2 >= 0] Move2(i, j) □
               [i - 1 >= 0 && j + 2 < N] Move3(i, j) □
               [i + 1 < N && j - 2 >= 0] Move4(i, j) □
               [i + 1 < N && j + 2 < N] Move5(i, j) □
               [i + 2 < N && j - 1 >= 0] Move6(i, j) □
               [i + 2 < N && j + 1 < N] Move7(i, j);
//each jump will update the board and counter
Move0(i, j) = [board[(i - 2) * N + j - 1] == 0] jump0{board[(i - 2) * N + j - 1] = 1;
               steps ++} → Knight(i - 2, j - 1);
Move1(i, j) = [board[(i - 2) * N + j + 1] == 0] jump1{board[(i - 2) * N + j + 1] = 1;
               steps ++} → Knight(i - 2, j + 1);
Move2(i, j) = [board[(i - 1) * N + j - 2] == 0] jump2{board[(i - 1) * N + j - 2] = 1;
               steps ++} → Knight(i - 1, j - 2);
Move3(i, j) = [board[(i - 1) * N + j + 2] == 0] jump3{board[(i - 1) * N + j + 2] = 1;
               steps ++} → Knight(i - 1, j + 2);
Move4(i, j) = [board[(i + 1) * N + j - 2] == 0] jump4{board[(i + 1) * N + j - 2] = 1;
               steps ++} → Knight(i + 1, j - 2);

```

```

Move5(i, j) = [board[(i + 1) * N + j + 2] == 0]jump5{board[(i + 1) * N + j + 2] = 1;
               steps ++} → Knight(i + 1, j + 2);
Move6(i, j) = [board[(i + 2) * N + j - 1] == 0]jump6{board[(i + 2) * N + j - 1] = 1;
               steps ++} → Knight(i + 2, j - 1);
Move7(i, j) = [board[(i + 2) * N + j + 1] == 0]jump7{board[(i + 2) * N + j + 1] = 1;
               steps ++} → Knight(i + 2, j + 1);
Game(i, j) = start{board[i * N + j] = 1} → Knight(i, j);
GameInstance = Game(0, 0);
//////////The Properties//////////
#define goal (board[0] == 1 && board[1] = 1 && board[2] = 1 && board[3] = 1
&& board[4] = 1 && board[5] = 1 && board[6] = 1 && board[7] = 1
&& board[8] = 1 && board[9] = 1 && board[10] = 1 && board[11] = 1
&& board[12] = 1 && board[13] = 1 && board[14] = 1 && board[15] = 1
&& board[16] = 1 && board[17] = 1 && board[18] = 1 && board[19] = 1
&& board[20] = 1 && board[21] = 1 && board[22] = 1 && board[23] = 1
&& board[24] = 1);
#assert GameInstance reaches goal;

```

A.6 The Tower of Hanoi Puzzle

The tower of Hanoi puzzle consists of three rods and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks neatly stacked in order of size on one rod, the smallest at the top, thus making a conical shape. The puzzle is solved when disks are moved to the target rod where disks are stacked in order of size. Information of disk order on each rod is modelled as global shared variables, for example, a shared array *column1* in our CSP# model.

```

//////////The Model//////////
#define N 6;
var column1[N + 1];
var column2[N + 1];
var column3[N + 1];
var size[4];
Init() = ini{size[1] = N; size[2] = 0; size[3] = 0; column1[1] = 6; column1[2] = 5;
             column1[3] = 4; column1[4] = 3; column1[5] = 2; column1[6] = 1; } → Skip;

```

```

Move1To2() = ifa((size[1] > 0 && column1[size[1]] < column2[size[2]]) ||
  (size[1] > 0 && size[2] == 0)) {
  move.1.2.column1[size[1]]{size[2] = size[2] + 1;
  column2[size[2]] = column1[size[1]]; size[1] = size[1] - 1; } → Skip };
Move1To3() = ifa((size[1] > 0 && column1[size[1]] < column3[size[3]]) ||
  (size[1] > 0 && size[3] == 0)) {
  move.1.3.column1[size[1]]{size[3] = size[3] + 1;
  column3[size[3]] = column1[size[1]]; size[1] = size[1] - 1; } → Skip };
Move2To1() = ifa((size[2] > 0 && column2[size[2]] < column1[size[1]]) ||
  (size[2] > 0 && size[1] == 0)) {
  move.2.1.column2[size[2]]{size[1] = size[1] + 1;
  column1[size[1]] = column2[size[2]]; size[2] = size[2] - 1; } → Skip };
Move2To3() = ifa((size[2] > 0 && column2[size[2]] < column3[size[3]]) ||
  (size[2] > 0 && size[3] == 0)) {
  move.2.3.column2[size[2]]{size[3] = size[3] + 1;
  column3[size[3]] = column2[size[2]]; size[2] = size[2] - 1; } → Skip };
Move3To1() = ifa((size[3] > 0 && column3[size[3]] < column1[size[1]]) ||
  (size[3] > 0 && size[1] == 0)) {
  move.3.1.column3[size[3]]{size[1] = size[1] + 1;
  column1[size[1]] = column3[size[3]]; size[3] = size[3] - 1; } → Skip };
Move3To2() = ifa((size[3] > 0 && column3[size[3]] < column2[size[2]]) ||
  (size[3] > 0 && size[2] == 0)) {
  move.3.2.column3[size[3]]{size[2] = size[2] + 1;
  column2[size[2]] = column3[size[3]]; size[3] = size[3] - 1; } → Skip };
Move() = Move1To2() □ Move1To3() □ Move2To1() □
  Move2To3() □ Move3To1() □ Move3To2();
System() = Init(); System1();
System1() = Move(); System1();
//////////The Properties//////////
#define goal (column3[1] == 6 && column3[2] == 5 &&
  column3[3] == 4 && column3[4] == 3 && column3[5] == 2 && column3[6] == 1);
#assert System() reaches goal;

```

A.7 Concurrent Stack

A concurrent stack allows multiple readers to access the shared variable at the same time, but only one writer to update the value; readers cannot access the shared variable when it is written.[41]. The verified property is trace refinement denoting that the trace of the

concrete stack implementation is a subset of the trace of the abstract one.

```

#define N 3; //number of processes
#define SIZE 2; //stack size
var H = 0; //shared head pointer for the concrete implementation
var HL[N]; //local variable to store the temporary head value
var HA = 0; //shared head pointer for the abstract implementation
var HLA[N]; //local variable to store the temporary head value
//////////The Concrete Implementation Model//////////
// Algorithm 2, procedure push
Push(i) = push_inv.i → τ{HL[i] = H; } → PushLoop(i);
PushLoop(i) =
  ifa (HL[i] == H) {
    τ{if (H < SIZE) {H = H + 1; } HL[i] = H; }
    → push_res.i.HL[i]{HL[i] = 0} → Skip
  } else {
    τ{HL[i] = H; } → PushLoop(i)
  };
// Algorithm 2, procedure pop
Pop(i) = pop_inv.i → τ{HL[i] = H; } → PopLoop(i);
PopLoop(i) =
  if (HL[i] == 0) {
    pop_res.i.0{HL[i] = 0} → Skip
  } else {
    (ifa (HL[i] != H) {
      τ{HL[i] = H; } → PopLoop(i)
    } else {
      τ{H = H - 1; HL[i] = H; } →
      pop_res.i.(HL[i] + 1){HL[i] = 0} → Skip
    })
  };
Process(i) = (Push(i) □ Pop(i)); Process(i);
Stack() = ||| x : {0..N - 1}@Process(x);
//////////The Abstract Specification Model//////////
// Algorithm 3, procedure push
PushAbs(i) = push_inv.i → τ{if (HA < SIZE) {HA = HA + 1; };
  HLA[i] = HA; } → push_res.i.HLA[i]{HLA[i] = 0; } → Skip;

```

```

// Algorithm 3, procedure pop
PopAbs(i) = pop_inv.i →
  (ifa(HA == 0) {
    τ{HLA[i] = -1} → pop_res.i.(HLA[i] + 1){HLA[i] = 0; } → Skip
  } else {
    τ{HA = HA - 1; HLA[i] = HA; } →
    pop_res.i.(HLA[i] + 1){HLA[i] = 0; } → Skip
  });
ProcessAbs(i) = (PushAbs(i) □ PopAbs(i)); ProcessAbs(i);
StackAbs() = ||| x : {0..N - 1}@ProcessAbs(x);
//////////The Properties//////////
#assert Stack() refines StackAbs();

```

A.8 Peterson's Algorithm

Peterson's algorithm [59], a concurrent programming algorithm, is designed for mutual exclusion that allows two processes to share a single-use resource without conflicts with only shared memory for communication. The algorithm can be generalized for more than two processes, as modelled in our experiment. The property is mutual exclusion, which is specified by reachability assertion in CSP# model.

```

//////////The Model//////////
#define N 3;
var step[N];
var pos[N];
var counter = 0; //which counts how many processes are in the critical session.
Process(i) = localupdate.i.1 → Repeat(i, 1); css.i{counter = counter + 1; } →
  csse.i{counter = counter - 1; } → reset{pos[i] = 0; } → Process(i);
Repeat(i, j) = [j < N] update.i.1{pos[i] = j; } → update.i.2{step[j] = i; } →
  atomic{localupdate.i.j.true → localupdate.i.j.true.0 → Skip};
  Repeat1(i, true, 0, j); atomic{localupdate.i.j + 1 → Skip};
  Repeat(i, j + 1)
  □ [j == N] Skip;

```

```

Repeat1(i, lb, k, j) = [k < N]ifa(lb && k != i){
    Repeat2(i, lb, k, j)
} else {
    Skip; atomic{localupdate.i.j.lb.k + 1 → Skip};
    Repeat1(i, lb, k + 1, j)
}
□ [k == N] Skip;
Repeat2(i, lb, k, j) = [lb && pos[k] >= pos[i]]
    (ifa(step[j] == i){
        atomic{localupdate.i.j.true.k → Skip}; Repeat2(i, true, k, j)
    } else {
        atomic{localupdate.0.j.false.k → Skip}; Repeat2(i, false, k, j)
    })
□ [!lb || pos[k] < pos[i]] Skip; atomic{localupdate.i.j.lb.k + 1 → Skip};
    Repeat1(i, lb, k + 1, j);
Peterson() = ||| i : {0..N - 1}@Process(i);
//////////The Properties//////////
#define goal counter > 1;
#assert Peterson() reaches goal;

```

Appendix B

Monotonicity of CSP# Process Combinators

This appendix presents the detailed proof of the monotonicity of the CSP# process constructs. Given any two processes P and Q such that $P \sqsupseteq Q$, then given any process R , the following auxiliary laws should be satisfied.

Law A.1

$$(P \wedge R) \sqsupseteq (Q \wedge R), \text{ provided that } P \sqsupseteq Q.$$

Proof:

$$\begin{aligned}
 & (P \wedge R) \sqsupseteq (Q \wedge R) && [\sqsupseteq] \\
 = & [(P \wedge R) \Rightarrow (Q \wedge R)] && [\text{propositional calculus}] \\
 = & [((P \wedge R) \Rightarrow Q) \wedge ((P \wedge R) \Rightarrow R)] && [\text{propositional calculus}] \\
 = & [((P \Rightarrow Q) \vee (R \Rightarrow Q)) \wedge ((P \Rightarrow R) \vee (R \Rightarrow R))] && [\text{assumption}] \\
 = & [(true \vee (R \Rightarrow Q)) \wedge ((P \Rightarrow R) \vee (R \Rightarrow R))] && [\text{propositional calculus}] \\
 = & [true \wedge true] && [\text{propositional calculus}] \\
 = & true && \square
 \end{aligned}$$

Law A.2

$$(P \vee R) \sqsupseteq (Q \vee R), \text{ provided that } P \sqsupseteq Q.$$

Proof:

$$\begin{aligned}
 & (P \vee R) \sqsupseteq (Q \vee R) && [\sqsupseteq] \\
 = & [(P \vee R) \Rightarrow (Q \vee R)] && [\text{propositional calculus}] \\
 = & [(P \Rightarrow (Q \vee R)) \wedge (R \Rightarrow (Q \vee R))] && [\text{propositional calculus}] \\
 = & [((P \Rightarrow Q) \vee (P \Rightarrow R)) \wedge ((R \Rightarrow Q) \vee (R \Rightarrow R))] && [\text{assumption}] \\
 = & [true \vee (P \Rightarrow R)) \wedge ((R \Rightarrow Q) \vee (R \Rightarrow R))] && [\text{propositional calculus}] \\
 = & [true \wedge true] && [\text{propositional calculus}] \\
 = & true && \square
 \end{aligned}$$

The CSP# sequential composition construct is monotonic (see **Law A.3** and **Law A.4**).

Law A.3

$$(P; R) \sqsupseteq (Q; R), \text{ provided that } P \sqsupseteq Q.$$

Proof:

$$\begin{aligned}
 & (P; R) \sqsupseteq (Q; R) && [\sqsupseteq] \\
 = & \forall obs, obs' \bullet ((P; R) \Rightarrow (Q; R))^1 && [4.1.3.5] \\
 = & \forall obs, obs' \bullet \left(\begin{array}{c} \exists obs_0 \bullet (P[obs_0/obs'] \wedge R[obs_0/obs]) \\ \Rightarrow \\ \exists obs_0 \bullet (Q[obs_0/obs'] \wedge R[obs_0/obs]) \end{array} \right) && \left[\begin{array}{c} \text{assumption, } \sqsupseteq \\ \text{and Lemma 2} \end{array} \right] \\
 = & true && \square
 \end{aligned}$$

Lemma 2. $\forall obs, obs' \bullet (\exists m \bullet (P(obs, m) \wedge R(m, obs')) \Rightarrow \exists m \bullet (Q(obs, m) \wedge R(m, obs')))$
holds, provided that $\forall obs, obs' \bullet (P(obs, obs') \Rightarrow Q(obs, obs'))$.

Proof:

¹The term *obs* represents the set of observational variables *ok*, *wait*, *tr*, as is the case of *obs'*.

1	$\forall obs, obs' \bullet (P(obs, obs') \Rightarrow Q(obs, obs'))$	premise
2	$obs_1 \quad \forall obs' \bullet (P(obs_1, obs') \Rightarrow Q(obs_1, obs'))$	$\forall obs \ e \ 1$
3	$obs'_1 \quad P(obs_1, obs'_1) \Rightarrow Q(obs_1, obs'_1)$	$\forall obs' \ e \ 2$
4	$\exists m \bullet (P(obs_1, m) \wedge R(m, obs'_1))$	assumption
5	$m_0 \quad P(obs_1, m_0) \wedge R(m_0, obs'_1)$	$\exists m \ e \ 4$
6	$P(obs_1, m_0) \Rightarrow Q(obs_1, m_0)$	$\forall obs' \ e \ 2$
7	$P(obs_1, m_0)$	$\wedge e_1 \ 5$
8	$Q(obs_1, m_0)$	$\Rightarrow e \ 6, 7$
9	$R(m_0, obs'_1)$	$\wedge e_2 \ 5$
10	$Q(obs_1, m_0) \wedge R(m_0, obs'_1)$	$\wedge i \ 8, 9$
11	$\exists m \bullet (Q(obs_1, m) \wedge R(m, obs'_1))$	$\exists m \ i \ 10$
12	$\exists m \bullet (Q(obs_1, m) \wedge R(m, obs'_1))$	$\exists m \ 4, 5 - 11$
13	$\exists m \bullet (P(obs_1, m) \wedge R(m, obs'_1)) \Rightarrow$ $\exists m \bullet (Q(obs_1, m) \wedge R(m, obs'_1))$	$\Rightarrow i \ 4 - 12$
14	$\forall obs' \bullet (\exists m \bullet (P(obs_1, m) \wedge R(m, obs'_1)) \Rightarrow$ $\exists m \bullet (Q(obs_1, m) \wedge R(m, obs'_1)))$	$\forall obs' \ i \ 3 - 13$
15	$\forall obs, obs' \bullet (\exists m \bullet (P(obs, m) \wedge R(m, obs'_1)) \Rightarrow$ $\exists m \bullet (Q(obs, m) \wedge R(m, obs'_1)))$	$\forall obs \ i \ 2 - 14$

Law A.4

$(R; P) \sqsupseteq (R; Q)$, provided that $P \sqsupseteq Q$.

Proof:

$$\begin{aligned}
 & (R; P) \sqsupseteq (R; Q) && [\sqsupseteq] \\
 = & \forall obs, obs' \bullet ((R; P) \Rightarrow (R; Q)) && [4.1.3.5] \\
 = & \forall obs, obs' \bullet \left(\begin{array}{c} \exists obs_0 \bullet (R[obs_0/obs'] \wedge P[obs_0/obs]) \\ \Rightarrow \\ \exists obs_0 \bullet (R[obs_0/obs'] \wedge Q[obs_0/obs]) \end{array} \right) \left[\begin{array}{c} \text{assumption, } \sqsupseteq \\ \text{and Lemma 3} \end{array} \right] \\
 = & true && \square
 \end{aligned}$$

Lemma 3. $\forall obs, obs' \bullet (\exists m \bullet (R(obs, m) \wedge P(m, obs')) \Rightarrow \exists m \bullet (R(obs, m) \wedge P(m, obs')))$
holds, provided that $\forall obs, obs' \bullet (P(obs, obs') \Rightarrow Q(obs, obs'))$.

Proof:

1	$\forall obs, obs' \bullet (P(obs, obs') \Rightarrow Q(obs, obs'))$	premise
2	$obs'_1 \quad \forall obs \bullet (P(obs, obs'_1) \Rightarrow Q(obs, obs'_1))$	$\forall obs' \ e \ 1$
3	$obs_1 \quad P(obs_1, obs'_1) \Rightarrow Q(obs_1, obs'_1)$	$\forall obs \ e \ 2$
4	$\exists m \bullet (R(obs_1, m) \wedge P(m, obs'_1))$	assumption
5	$m_0 \quad R(obs_1, m_0) \wedge P(m_0, obs'_1)$	$\exists m \ e \ 4$
6	$P(m_0, obs'_1) \Rightarrow Q(m_0, obs'_1)$	$\forall obs \ e \ 2$
7	$P(m_0, obs'_1)$	$\wedge e_2 \ 5$
8	$Q(m_0, obs'_1)$	$\Rightarrow e \ 6, 7$
9	$R(obs_1, m_0)$	$\wedge e_1 \ 5$
10	$R(obs_1, m_0) \wedge Q(m_0, obs'_1)$	$\wedge i \ 8, 9$
11	$\exists m \bullet (R(obs_1, m) \wedge Q(m, obs'_1))$	$\exists m \ i \ 10$
12	$\exists m \bullet (R(obs_1, m) \wedge Q(m, obs'_1))$	$\exists m \ 4, 5 - 11$
13	$\exists m \bullet (R(obs_1, m) \wedge P(m, obs'_1)) \Rightarrow$ $\exists m \bullet (R(obs_1, m) \wedge Q(m, obs'_1))$	$\Rightarrow i \ 4 - 12$
14	$\forall obs \bullet (\exists m \bullet (R(obs, m) \wedge P(m, obs'_1)) \Rightarrow$ $\exists m \bullet (R(obs, m) \wedge Q(m, obs'_1)))$	$\forall obs \ i \ 3 - 13$
15	$\forall obs, obs' \bullet (\exists m \bullet (R(obs, m) \wedge P(m, obs')) \Rightarrow$ $\exists m \bullet (R(obs, m) \wedge Q(m, obs')))$	$\forall obs' \ i \ 2 - 14$

Synchronous output/input is monotonic (see **Law A.5** and **Law A.6**).

Law A.5

$(ch!exp \rightarrow P) \sqsupseteq (ch!exp \rightarrow Q)$, provided that $P \sqsupseteq Q$.

Proof:

$$\begin{aligned}
 & (ch!exp \rightarrow P) \\
 = & \mathbf{H} \left(ok' \wedge \left(\begin{array}{l} ch? \notin ref' \wedge tr' = tr \\ \triangleleft wait' \triangleright \\ \exists s \in \mathbf{S} \bullet tr' = tr \cap \langle (s, ch!\mathcal{A}[[exp]](s)) \rangle \end{array} \right) \right) ; P \quad \begin{array}{l} [4.1.3.3] \\ \left[\begin{array}{l} \text{assum-} \\ \text{ption} \\ \text{and A.4} \end{array} \right] \end{array} \\
 \sqsubseteq & \mathbf{H} \left(ok' \wedge \left(\begin{array}{l} ch? \notin ref' \wedge tr' = tr \\ \triangleleft wait' \triangleright \\ \exists s \in \mathbf{S} \bullet tr' = tr \cap \langle (s, ch!\mathcal{A}[[exp]](s)) \rangle \end{array} \right) \right) ; Q \quad [4.1.3.3] \\
 = & ch!exp \rightarrow Q \quad \square
 \end{aligned}$$

Law A.6

$$(ch?m \rightarrow P(m)) \sqsubseteq (ch?m \rightarrow Q(m)), \text{ provided that } \forall m \in \mathbf{T} \bullet P(m) \sqsubseteq Q(m).$$

Proof:

$$\begin{aligned}
 & ch?m \rightarrow P(m) \\
 = & \exists v \in \mathbf{T} \bullet \mathbf{H} \left(ok' \wedge \left(\begin{array}{l} ch! \notin ref' \wedge tr' = tr \\ \triangleleft wait' \triangleright \\ tr' = tr \cap \langle (s, ch?v) \rangle \end{array} \right) \right) ; P(v) \quad \begin{array}{l} [4.1.3.3] \\ \left[\begin{array}{l} \text{assumption,} \\ \text{A.4, and predi-} \\ \text{cate calculus} \end{array} \right] \end{array} \\
 \sqsubseteq & \exists v \in \mathbf{T} \bullet \mathbf{H} \left(ok' \wedge \left(\begin{array}{l} ch! \notin ref' \wedge tr' = tr \\ \triangleleft wait' \triangleright \\ tr' = tr \cap \langle (s, ch?v) \rangle \end{array} \right) \right) ; Q(v) \quad [4.1.3.3] \\
 = & ch?m \rightarrow Q(m) \quad \square
 \end{aligned}$$

The CSP# data operation prefixing construct is monotonic (see **Law A.7**).

Law A.7

$(e\{prog\} \rightarrow P) \sqsupseteq (e\{prog\} \rightarrow Q)$, *provided that* $P \sqsupseteq Q$.

Proof:

$$\begin{aligned}
 & e\{prog\} \rightarrow P \tag{4.1.3.4} \\
 = & \mathbf{H} \left(\begin{array}{l} ok' \wedge \left(\begin{array}{l} \exists s \in \mathbf{S} \bullet (tr' = tr \frown \langle (s, \perp) \rangle \wedge (s, \perp) \in \mathcal{C}[\![prog]\!]) \\ \triangleleft wait' \triangleright \\ \exists s, s' \in \mathbf{S} \bullet (tr' = tr \frown \langle (s, s') \rangle \wedge (s, s') \in \mathcal{C}[\![prog]\!]) \\ \wedge (s, \perp) \notin \mathcal{C}[\![prog]\!] \end{array} \right) \end{array} \right); P \\
 & \tag{assumption and A.4} \\
 \sqsupseteq & \mathbf{H} \left(\begin{array}{l} ok' \wedge \left(\begin{array}{l} \exists s \in \mathbf{S} \bullet (tr' = tr \frown \langle (s, \perp) \rangle \wedge (s, \perp) \in \mathcal{C}[\![prog]\!]) \\ \triangleleft wait' \triangleright \\ \exists s, s' \in \mathbf{S} \bullet (tr' = tr \frown \langle (s, s') \rangle \wedge (s, s') \in \mathcal{C}[\![prog]\!]) \\ \wedge (s, \perp) \notin \mathcal{C}[\![prog]\!] \end{array} \right) \end{array} \right); Q \tag{4.1.3.4} \\
 = & e\{prog\} \rightarrow Q \tag{\square}
 \end{aligned}$$

The CSP# state guard is monotonic (see **Law A.8**).

Law A.8

$[b]P \sqsupseteq [b]Q$, *provided that* $P \sqsupseteq Q$.

Proof:

$$\begin{aligned}
 & [b]P \sqsupseteq [b]Q \quad [4.1.3.7 \text{ and } \sqsupseteq] \\
 = & \left[\begin{array}{c} P \triangleleft \mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr' \triangleright \text{Stop} \\ \Rightarrow \\ Q \triangleleft \mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr' \triangleright \text{Stop} \end{array} \right] \\
 & \quad [predicate \text{ calculus}] \\
 = & \left[\begin{array}{c} \left(\begin{array}{c} (P \wedge \mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \\ \Rightarrow \\ (Q \wedge \mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \end{array} \right) \\ \vee \\ \left(\begin{array}{c} (P \wedge \mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \\ \Rightarrow \\ (\text{Stop} \wedge \neg(\mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr')) \end{array} \right) \\ \wedge \\ \left(\begin{array}{c} (\text{Stop} \wedge \neg(\mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr')) \\ \Rightarrow \\ (Q \wedge \mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \end{array} \right) \\ \vee \\ \left(\begin{array}{c} (\text{Stop} \wedge \neg(\mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr')) \\ \Rightarrow \\ (\text{Stop} \wedge \neg(\mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr')) \end{array} \right) \end{array} \right] \\
 & \quad [predicate \text{ calculus}]
 \end{aligned}$$

$$\begin{aligned}
 &= \left[\left(\begin{array}{c} (P \Rightarrow Q) \\ \vee \\ ((\mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \Rightarrow Q) \\ \vee \\ \left(\begin{array}{c} (P \wedge \mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \\ \Rightarrow \\ (Stop \wedge \neg(\mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr')) \end{array} \right) \end{array} \right) \right] \\
 &\quad \wedge \left[\left(\begin{array}{c} (Stop \wedge \neg(\mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr')) \\ \Rightarrow \\ (Q \wedge \mathcal{B}(b)(\pi_1(\text{head}(tr' - tr))) = \text{true} \wedge tr < tr') \end{array} \right) \right] \\
 &\quad \vee \\
 &\quad \left(\text{true} \right) \\
 &\hspace{15em} [\textit{assumption and predicate calculus}] \\
 &= \text{true} \wedge \text{true} \hspace{15em} [\textit{predicate calculus}] \\
 &= \text{true} \hspace{15em} \square
 \end{aligned}$$

The CSP# parallel composition is monotonic (see **Law A.9** and **Law A.10**).

Law A.9

$$P \parallel R \sqsupseteq Q \parallel R$$

provided that $P \sqsupseteq Q$, and common actions and synchronous channel outputs/inputs of the two parallel processes (denoted as set X) are the same.

Proof:

$$\begin{aligned}
 & P \sqsupseteq Q && [\sqsupseteq] \\
 = & [P \Rightarrow Q] && [\text{predicate calculus}] \\
 = & [P[0.\text{obs}/\text{obs}] \Rightarrow Q[0.\text{obs}/\text{obs}]] && [\sqsupseteq] \\
 = & P[0.\text{obs}/\text{obs}] \sqsupseteq Q[0.\text{obs}/\text{obs}] && [\text{Law A.1}] \\
 & (P[0.\text{obs}/\text{obs}] \wedge R[1.\text{obs}/\text{obs}]) \\
 = & \sqsupseteq && [\text{Law A.3}] \\
 & (Q[0.\text{obs}/\text{obs}] \wedge R[1.\text{obs}/\text{obs}]) \\
 & (P[0.\text{obs}/\text{obs}] \wedge R[1.\text{obs}/\text{obs}]); M(X) \\
 \Rightarrow & \sqsupseteq && [4.1.3.8] \\
 & (Q[0.\text{obs}/\text{obs}] \wedge R[1.\text{obs}/\text{obs}]); M(X) \\
 = & P \parallel R \sqsupseteq Q \parallel R && \square
 \end{aligned}$$

Law A.10

$$R \parallel P \sqsupseteq R \parallel Q,$$

provided that $P \sqsupseteq Q$, and common actions and synchronous channel outputs/inputs of the two parallel processes (denoted as set X) are the same.

Proof:

$$\begin{aligned}
 & P \sqsupseteq Q && [\sqsupseteq] \\
 = & [P \Rightarrow Q] && [\text{predicate calculus}] \\
 = & [P[1.\text{obs}/\text{obs}] \Rightarrow Q[1.\text{obs}/\text{obs}]] && [\sqsupseteq] \\
 = & P[1.\text{obs}/\text{obs}] \sqsupseteq Q[1.\text{obs}/\text{obs}] && [A.1] \\
 & (P[1.\text{obs}/\text{obs}] \wedge R[0.\text{obs}/\text{obs}]) \\
 = & \sqsupseteq && [\text{predicate calculus}] \\
 & (Q[1.\text{obs}/\text{obs}] \wedge R[0.\text{obs}/\text{obs}]) \\
 & (R[0.\text{obs}/\text{obs}] \wedge P[1.\text{obs}/\text{obs}]) \\
 = & \sqsupseteq && [A.3] \\
 & (R[0.\text{obs}/\text{obs}] \wedge Q[1.\text{obs}/\text{obs}]) \\
 & (R[0.\text{obs}/\text{obs}] \wedge P[1.\text{obs}/\text{obs}]); M(X) \\
 \Rightarrow & \sqsupseteq && [4.1.3.8] \\
 & (R[0.\text{obs}/\text{obs}] \wedge Q[1.\text{obs}/\text{obs}]); M(X) \\
 = & R \parallel P \sqsupseteq R \parallel Q && \square
 \end{aligned}$$

Since the semantics of other CSP# processes (i.e., event prefixing, external/internal choice and recursion) is the same as that of CSP, the proof is omitted here.