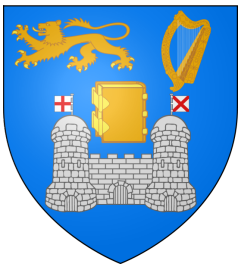
prioritised slotted-*Circus*

Paweł Gancarski

September 13, 2012



A thesis submitted to the University of Dublin, Trinity Collage
in fulfilment of the requirements for the degree of
Doctor of Philosophy (Computer Science).

b

Abstract

The main difference between a software and hardware design is that the hardware cannot be patched or updated after releasing. Also in the case of software, compilation and tests can be performed with the "click of a button", whereas with hardware it often implies the creation of a very expensive prototype. In spite of all the differences, hardware designing methodology seems to be going in the direction of software. The reason for it might lay in the increasing size and complexity of hardware solutions, or the fact that FPGA technology allows rapid hardware prototyping and re-configuration. This work presents slotted-*Circus* , a process algebra based on *Circus* with the added notion of time, and its prioritised version prioritised slotted-*Circus*. The dependence between the restrictions on expressiveness of the priority, safety, and complexity of its behaviour is investigated. Finally, how prioritised slotted-*Circus* can be used for verification of Wireless Sensor Networks algorithms and in the area of the real time operating systems is presented.

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidate's own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Paweł Gancarski

Summary

The work presented in this thesis can be divided into two main parts, the work on the denotational semantics of slotted-*Circus* and the work on a number of aspects of priority. The slotted-*Circus* denotational semantics is a continuation of the research started by a number of people. Details of the authors' contribution in that aspect are summarised in section 3.1.The results include detection and a fix of the state visibility issue (Sec. 3.6.9) published in [BGW09], and complete slotted-*Circus* denotational semantics along with a number of proofs published in [BG09]and [GB09].

The second part of this work is focused on the notion of priority. A prioritised version of slotted-*Circus* has been defined and published [GB10]. In chapter 4 the idea of immediate causality and untimed communication is investigated, presenting ways of dealing with the side effects that they introduce (Sec. 4.4) and providing arguments for their usefulness. The work presented herein provides both informal (Sec. 5.2) and formal (Sec. 5.4) arguments against attempts to define priority in the untimed frameworks like CSP, Ada, or occam. In chapter 6 the laws of prioritised slotted-*Circus* are investigated focusing on handling priority. Finally the possibilities that are introduced by priority are presented in the case study in chapter 7. This work is focused on presenting a wide range of possible use cases rather than trying to prove usefulness of priority on one example. This is especially important because the possible uses range from real time hardware implementation, specification and verification of wireless sensors networks algorithms to real time operating systems.

The search for an appropriate case study to present the idea of priority resulted in a contribution to a paper in the area of wireless sensors networks that is currently awaiting publication. Results presented in the paper are not included in this work.

Acknowledgements

Most of all I would like to express my gratitude to my parents who supported me throughout my adventurous studies in the cold and rainy island of Ireland. I would also like to give my thanks to Miren, for her constant support and the“delicate” pressure inflicted on me to keep working.

Finally I would like to acknowledge the time and effort of Andrew Butterfield which were sacrificed on this work far above responsibilities of a supervisor. Our numerous discussions were invaluable to the achievement of the results presented here.

Glossary of notation

Logic		
$true$	the logical true	
$false$	the logical false	
$=$	equality	
$P \equiv Q$	equivalence, **iff**, $[P = Q]$	
$\widehat{=}$	is defined by	
$=_{df}$	is defined by	
iff	if and only if	
$P \vee Q$	P or Q, disjunction	
$P \wedge Q$	P and Q, conjunction	
$\neg P$	not P	
$P \Rightarrow Q$	if P then Q	
$\exists x \bullet P$	there exists x such that P	
$\forall x \bullet P$	for all x, P	
$[]$	universal closure	[def:closure]:p. 10 footnote
$\{ x \bullet P(x) \}$	set of all x such that $P(x)$ holds	
Operators		
$;$	Sequential composition	2.1.2, [Seq:def]:p. 33
$\lhd c \rhd$	conditional choice	2.1.2, [Cond:def]:p. 33
$\sqcap$	nondeterministic choice	2.1.2, [NDet:def]:p. 33
$\bigsqcap_{i \in I}$	parameterised nondeterministic choice	2.1.2
$\sqsubseteq$	refinement	2.1.2
Simple actions		
$I\!I$	Skip, identity for sequential composition	2.1.2
$I\!I_R$	reactive Skip, identity for sequential composition in slotted-*Circus*	[Ilr:def]:p. 26
Functions		
$\circ$	function composition	[def:composition]:p. 10 footnote
$F : S \rightarrow T$	F is a total mapping from S to T	
Relations		
$\leq$	partial order	2.1.4
$\bigwedge$	greatest lower bound,infimum , meet	2.1.4
$\bigvee$	least upper bound, supremum, join	2.1.4
μ	least fixed point	2.1.4
$x \mapsto y$	x maps to y	
$\leftrightarrow$	relation type	

Low level slotted-*Circus*

$\sharp S$	length of S	
$\#\!\!\#$	slots addition operator	[CAT:def]:p. 109
$\backslash\backslash$	slots subtraction operator	[DF:binop]:p. 110
$eqvref(ss)$	function extracting the last refusals set from slots	[ER:def]:p. 107
$\{\mid channel^+ \mid\}$	set of all possible channels with a name starting from $channel$.	

Other

λ	lambda expressions
$state \oplus \{x \mapsto value\}$	update the value of x within $state$ with $value$
$val(e, state)$	value of an expression e in the $state$ environment
$\mathbb{P}$	power set
s^*	sequence of s
s^+	non empty sequence of s, $s^+ = s, s^*$

Contents

Chapter 1

Introduction

The main difference between a software and hardware design is that the hardware cannot be patched or updated after releasing. Also in the case of software, compilation and tests can be performed with the "click of the button", whereas with hardware it often implies the creation of a very expensive prototype. In spite of all the differences, hardware designing methodology seems to be going in the direction of software [BW06a] [Sal03]. The reason for it might lay in the increasing size and complexity of hardware solutions, or the fact that FPGA technology allows rapid hardware prototyping and re-configuration. This work we proposes an adapted software verification method, that can help with creation and verification of hardware designs. The usability of proposed solution for verification of Wireless Sensor Networks algorithms and in the area of the real time operating systems ia also investigated.

1.1 Synchronous Hardware

Synchronous hardware is a type of digital circuit where an external clock signal is used to synchronise the execution of its parts. When given data on input, every logical unit needs to produce an output within the time controlled by the clock.

Because current software creation methods do not provide enough dependability, hardware manufacturers usually use solutions based on the theory of finite state machines, in order to verify the synchronous hardware. Some of the most successful examples are Lustre [HCRP91] used by Airbus and Forte [SJO+05] used by Intel.

The main problem with the finite state machines is the state explosion issue which appears in larger designs [CGJ+01]. The possible solutions are process algebras which provide a range of algebraic laws, allowing analysis and reasoning about equivalence or refinement. The work presented herein is based on one of the available process calculi - CSP [Hoa78].

1.2 GC6

Grand Challenges in Computing had their beginning in 2004 in Newcastle during the GCC04 conference. Grand challenges are characterised by tough problems requiring significant effort from a number of organisations which are also expected to recognise goals one or two decades in advance [HM05].

One of the challenges addressed by the wide community was GC6: Dependable systems evolution. GC6's challenge is to find a better way of creating software that can be described as serviceable, sound and secure [HM05]. A great number of tasks is required to be addressed in order to succeed. First, the GC6 community will need to develop integrated, sound and general theories to reason about problems; then adopt those theories and specialise them to a wide range of problems that computer scientists have to face; create tool support for those theories; and teach specialists how to use the provided tools. Finally, GC6 community will need to convince industry that using those methods is cost-effective. When the research on the challenge starts giving results, it

is suggested that a verified software repository should be created in order to increase software re-usability and reliability.

GC6 is a UK initiative which participates in related international initiatives like the Verified Software Initiative [HLMS07] or the Verified Software Repository. The way of addressing the problem of synchronous hardware verification chosen for this research is based on Unified Theories of Programming [HH98] which is one of the most active branches of GC6.

1.3 Handel-C

Handel-C [Cel05] is a subset of C language with channel based communication originating from CSP [Hoa85a]. It provides precise timing, parallel and sequential constructs, global variable assignments and the ability to describe multiple clock systems (asynchronously connected). It is specialised in the development of software solutions for the field programmable gate arrays (FPGA-s). Handel-C was developed and supported by Celoxica - a UK based company which origins can be traced back to the Hardware Compilation Group at Oxford University Computing Laboratory. In January 2009, Handel-C was purchased by Mentor Graphics.

Because of its similarities to CSP and the fact that Handel-C programs can be easily implemented and tested on FPGA, Handel-C is often chosen by researchers as a target for the code generation [RGGR], [IS07], [GR05], [PS]. This fact has encouraged an extensive research on the formalisation of the Handel-C semantics [BW06b], [PW08]. One of the main difficulties in modelling the behaviour of Handel-C is the `prialt` operator. `prialt` (prioritised alternation) is the implementation of an external choice operator from CSP. Because the nondeterministic behaviour is undesired in hardware (in fact, nondeterministic behaviour of hardware usually signifies its failure), `prialt` is removing the nondeterminism from the external choice by presenting the available options in a prioritised list. While the description of `prialt` is very precise and there can be no doubt on predicting its behaviour it proved very difficult to model [BW02], [BW05a].

"In the broader arena of process algebras in general, there has been considerable work done on priorities [CH88],[CNSL96], [CLN]. In [CLN] there is an overview of this area, but there is no close match between any of the priority schemes described therein, and that which features in the semantics of Handel-C." [BW05b]

This work defines prioritised choice operator that suits the behaviour of the `prialt` construct perfectly, and adds it to slotted-*Circus*, a language designed for reasoning about parallel time critical systems.

1.4 Wireless Networks

During the research on hardware designing and verification, and thanks to interesting results in the field of priority, it became clear that those results could prove to be very useful in the field of wireless sensor networks.

Wireless sensor networks (WSNs) are currently a heavily supported research area. Because of the parallelism, often dynamic environment and the need for precise synchronisation that reduce the power consumption, the behaviour of a WSN can be very complex [Aky06]. For that reason many attempts to model and verify WSN-s have been made [SG08], [SLM+09]. The model of time and parallelism available in the slotted-*Circus*, along with the priority introduced in this work was used to reason about the most common type of communication in WSNs - asynchronous broadcasting, which in result gives us ability to model and reason about the networks behaviour.

1.5 Existing tool support

There is currently a wide variety of tools available for software analysis and verification. While non of them is being used in this work it is still worth mentioning the solutions that are currently used in the similar areas of research. The most well known tool in the CSP community is FDR. FDR (Failures-Divergences Refinement) is

a refinement checker able to transform two CSP-M (machine readable version of CSP) processes into two corresponding Labelled Transition Systems (LTSs), and than to determine wether the refinement relation between them holds. Worth mentioning is that FDR is able to analyse CSP specification with a "press of the button". Also latest improvements introduced the ability to handle real-time specifications [PAW12]. Because it is possible to translate the CSP or CSP-based language [oai06] to Z specification [Spi92], it is possible to successfully use tool like Z-Eves, ProofPower or Isabelle/HOL. Finally worth mentioning is PAT (Process Analysis Toolkit) [SLD08], a very ambitious and fast growing set of tools that supports concurrent and real-time systems.

1.6 Research Question

This work we will explores the methods used by the UTP branch of GC6 researchers - *Circus* like languages [OCW07] and a family of unified theories. Adapting a *Circus* like language to fit synchronous hardware designs has been studied along with its other possible uses.

The idea for this research is to use a hardware description language (HDL) similar to one of the widely known programming languages, and use the presented theory to reason about it. The language chosen for this purpose is Handel-C [Cel05]. The language used to reason about the designs is an adapted version of slotted-*Circus* [BSW07].

Because Handel-C implements a notion of priority, the theory behind it have been investigated in order to model it in slotted-*Circus*. Once the prioritised version of slotted-*Circus* is defined, the dependence between the restrictions on expressiveness of the priority, safety, and complexity of its behaviour, will be determined.

Finally, the foundations for a formal method for synchronous hardware development will be laid. The solution should allow designing with different levels of abstraction, as well as providing a starting point for future verification and code generation.

1.7 Contribution

The work presented herein can be divided into two sections. Firstly, it presents the UTP denotational semantics of slotted-Circus (Chapter 3)- process algebra based on Circus with the added notion of time. The slotted-*Circus* part of this thesis is fairly complete and the strong contribution of other works needs to be mentioned (for details see section 3.1). The results include detection and a fix of the state visibility issue (Sec. 3.6.9) published in [BGW09], and complete slotted-*Circus* denotational semantics along with a number of proofs published in [BG09]and [GB09].

The second part of this work is about priority and introduces a new algebraic operator - prioritised choice - a deterministic implementation of the external choice. prioritised slotted-*Circus* has been defined and published in [GB10]. In chapter 4 the idea of immediate causality and untimed communication has been investigated, introducing ways of dealing with side effects that they introduce (Sec. 4.4) and giving arguments for their usefulness. The work presented herein provides both informal (Sec. 5.2) and formal (Sec. 5.4) arguments against attempts to define priority in the untimed frameworks like CSP, Ada, or occam. In chapter 6 the laws of prioritised slotted-*Circus* focusing on handling of priority have been investigated. Finally the possibilities that are introduced by priority are presented in the case study in chapter 7. This work is focused on presenting a wide range of possible use cases rather than trying to prove usefulness of priority on one example. This is especially important because the possible uses range from real time hardware implementation, specification, and verification of wireless sensors networks algorithms to real time operating systems.

Chapter 2

Background

2.1 UTP

2.1.1 Introduction

The semantic model of slotted-*Circus* is based on the Unifying Theories of Programming (UTP) [HH98], a mathematical framework for describing and connecting different theories. UTP allows us to give a full and precise meaning to language constructs in a unified way. Once defined, the language constructs can be mathematically verified against a list of desired properties as well as analysed in order to find new ones. Finally, UTP allows us to compare different theories and create links between them expressed using Galois connections. This way we can show and prove refinement or equality of two programs/specifications described with different languages.

2.1.2 Basic Examples

UTP models the behaviour of a language construct as a predicate relating its before- and after-state. State is recorded as two sets of variables (obs and obs'). First one (obs) stands for before-state and uses undashed variables e.g. $trace, wait, ok, state, ref$, while a set of after-state variables (obs') uses dashed variables e.g. $trace', wait', ok', state', ref'$. It is important to note that every dashed variable has its undashed counterpart and that the free variables in the definitions can only belong to those two sets (the alphabet is restricted to the finite lists of dashed and undashed variables). In almost all cases there are some basic operators common to every theory and extra care is always taken to maintain those definitions unchanged:

$$
\begin{aligned}
P;Q &=_{def} \exists obs_m \bullet P[obs_m/obs'] \wedge Q[obs_m/obs] \\
P \lhd c \rhd Q &=_{def} c \wedge P \vee \neg c \wedge Q \\
P \sqcap Q &=_{def} P \vee Q \\
\bigsqcap_{i \in I} P_i &=_{def} \exists i \in I \bullet P_i \\
Q \sqsubseteq P &=_{def} [P \Rightarrow Q] \\
\mathit{II} &=_{def} obs' = obs
\end{aligned}
$$

The first one - sequential composition ($P;Q$) - is the best one to describe the UTP idea of before and after state. In here the state in which process P finishes is the same as the beginning state of process Q. In order to do that a new set of observations is introduced (obs_m) which corresponds to a set of observation variables indexed with m e.g. $trace_m, wait_m, ok_m, state_m, ref_m$. Then simultaneous substitution is used to replace dashed variables of P and undashed of Q with obs_m. Finally obs_m is bounded with the existential quantifier, reducing the set of free variables of $P;Q$, to obs from P and obs' from Q.

The conditional $P \lhd c \rhd Q$ is used as a shorthand notation for IF (c) THEN (P) ELSE (Q). The properties of the definition can easily be checked by proving:

$$P \lhd true \rhd Q = P$$

$$P \lhd false \rhd Q = Q$$

Nondeterminism between two predicates $P \sqcap Q$ is defined as a logical disjunction. The definition can be extended to a nondeterministic choice over an indexed set $(\bigsqcap_{i \in I} P_i)$ using basic logical properties of disjunction.

UTP strongly supports the notion of refinement ($\sqsubseteq$). Refinement allows us to express that one process is an implementation of another. A good example of the use of refinement would be a proof that some kind of complicated device is compliant with a predefined standard ($DEV \sqsupseteq STANDARD$). In UTP this is defined as an implication quantified over all free variables (*obs* and *obs*′). $[DEV \Rightarrow STANDARD]$ [1] means that any behaviour accepted by *DEV* is an accepted behaviour of *STANDARD*. Intuitively helpful and very straightforward to prove can be the following property of nondeterminism:

$$(P \sqcap Q) \sqsubseteq P$$

2.1.3 Healthiness Conditions

By following the idea of "programs as predicates" [Hoa85b] in UTP, language syntax is viewed as a shorthand notation for the corresponding semantics, described using alphabetised predicates. This approach allows us to prove properties of a language, by proving that a property holds for any predicate over free, observational variables. Having freedom to write any predicate over observation variables often allows description of some unimplementable or senseless programs. In UTP this problem is solved by introduction of a concept of predicate/program "healthiness". A class of "healthy" predicates is restricted using a number of "healthiness conditions" that describe when a process is practical, feasible and desirable.

While in different theories healthiness conditions might vary, there are examples that are common to most of them. The common example of healthiness conditions that are used for most of the theories is that you can not change the past (trace can only grow). Another states that you can not work before being started (if the previous process is still working then we behave like identity). A healthiness condition might also demand that that a process's behaviour is independent of a past history, otherwise its implementation would need an unlimited memory to remember the past. In UTP healthiness conditions are assured by the use of predicate-transformers. The predicate transformers, for convenience, often overload the terminology "condition" and are called healthiness conditions.

$$\mathbf{H} : Pred \rightarrow Pred$$

Healthiness conditions can transform any unhealthy predicate into a healthy one, while leaving the healthy one unchanged. This property demands that they will be idempotent.[2]

$$\mathbf{H} \circ \mathbf{H} = \mathbf{H}$$

This idempotence property allows us to test a predicate for healthiness simply by checking if it is a fixed point of the transformer.

$$\mathbf{H}(P) = P$$

Typically in a theory there is more then one healthiness conditions. For that reason it is desirable that they commute.

$$\mathbf{Ha} \circ \mathbf{Hb} = \mathbf{Hb} \circ \mathbf{Ha}$$

[1] $[P]$ denotes the universal closure of P. $[P] = \forall free(P) \bullet P$, where $free(P)$ is a set of all the free variables of P

[2] $\circ$ denotes a standard function composition, $(f \circ g)(x) = f(g(x))$

This property allows us to pull a desired healthiness property from a healthy process whenever it's needed to complete a proof:

$$P = \mathbf{H_i}(P) \qquad for\ P\ \mathbf{H_{1-n}} healthy$$

It also makes it unnecessary to remember an order in which the healthiness conditions are applied to a healthy process. An example of healthiness that is common to most of the UTP defined languages is a reactive process. In the UTP book a process P is called reactive iff it is a fixed point of $\mathbf{R} =_{df} \mathbf{R}3 \circ \mathbf{R}2 \circ \mathbf{R}1$ where

$$\begin{aligned} \mathbf{R}1(P) &=_{df} P \wedge (tr \leq tr') \\ \mathbf{R}2(P) &=_{df} \sqcap_{s} P[s, s^\frown(tr' - tr)/tr, tr'] \\ \mathbf{R}3(P) &=_{df} I\!I \lhd wait \rhd P \end{aligned}$$

In here tr and tr' are an observation variables representing trace and *wait* is a boolean variable describing if a process has started. $\mathbf{R}1$ ensures that a process can only add events to the trace and it intuitively disallows time travelling or changing the past of a process. $\mathbf{R}2$ describes a healthiness condition mentioned before, that a process is not influenced by the events that appeared in the past. A fact to note at this point is that a process can still be influenced by the environment of the process or by other possible observation variables that represent state in which the process starts. Any process that is not $\mathbf{R}2$ healthy and so its behaviour varies for different histories ($tr' - tr$ varies for different tr) will be weakened by $\mathbf{R}2$. Finally $\mathbf{R}3$ makes sure that sequentially composed processes are well behaved and can not make any changes until a process running before them terminates (is not waiting - $\neg wait'$).

$$(P \wedge wait'); \mathbf{R}3(Q); \mathbf{R}3(S) = (P \wedge wait'); I\!I; I\!I = (P \wedge wait')$$

Healthiness condition describe fundamental rules for the language. They define the type of the language rather than the language itself by providing a sandbox in which the language constructs operate. While working with a new theory healthiness conditions should be treated with extra care. Every, even the smallest, change in their definition should be carefully analysed, and should be introduced only when necessary. Changes to healthiness condition should entail a re-check of all other definitions and proofs in the language.

2.1.4 Recursion in UTP

Recursion in UTP is defined as a weakest fixed point of a function on a predicate. In order to be able to understand how it works and what are its restrictions it is necessary to understand the meaning of complete lattice, monotonicity of UTP operators, theory of fixed points and finally Tarski's theorem [Tar55].

Complete Lattice

As mentioned before UTP has strong support for the notion of refinement. After taking a closer look into the definition of refinement it is possible to notice that $\sqsubseteq$ is reflexive, antisymmetric and transitive, and so is a partial order.

$$\begin{aligned} P \sqsupseteq P &= [P \Rightarrow P] = true \\ (P \sqsupseteq Q) \wedge (Q \sqsupseteq P) &= [P \Rightarrow Q] \wedge [Q \Rightarrow P] = [P \Leftrightarrow Q] = (P = Q) \\ (P \sqsupseteq Q) \wedge (Q \sqsupseteq R) &= ([P \Rightarrow Q] \wedge [Q \Rightarrow R]) \Rightarrow ([P \Rightarrow R]) = (P \sqsupseteq R) \end{aligned}$$

A partially ordered set $(L, \leq)$ is a complete lattice if every subset A of L has both a greatest lower bound $\bigwedge A$ (the infimum, also called the meet) and a least upper bound $\bigvee A$(the supremum, also called the join) in $(L, \leq)$. An important aspect of refinement is that together with the set of all possible first-order predicates it creates a complete lattice $(\mathbb{L}, \sqsubseteq)$. For any $\mathbb{A} \subseteq \mathbb{L}$, $\mathbb{A}$ has both meet and join, where meet can be defined as disjunction

over elements of $\mathbb{A}$: $\bigwedge \mathbb{A} = \underset{a \in \mathbb{A}}{\vee} a$, and join as a conjunction over elements of $\mathbb{A}$: $\bigvee \mathbb{A} = \underset{a \in \mathbb{A}}{\wedge} a$.

Tarski's Fixed Point Theorem

Theorem 1: *Let $(L, \leq)$ be any complete lattice. Suppose $f : L \to L$ is monotonic (or isotone), i.e., for all x, y in L, $x \leq y$ implies $f(x) \leq f(y)$. Then the set of all fixed points of f is a complete lattice with respect to $\leq$ [Tar55].*

Tarski's theorem is a key concept for the understanding of the most advanced aspects of UTP defined theories. In order to see how the theorem can be used for lattices with the refinement ($\sqsubseteq$) ordering, it is necessary to understand when the mentioned function $f : L \to L$ is monotonic under refinement ordering.

For any predicate S, P and Q the following are theorems:

$$P \sqsubseteq Q \Rightarrow (P \vee S) \sqsubseteq (Q \vee S)$$
$$P \sqsubseteq Q \Rightarrow (P \wedge S) \sqsubseteq (Q \wedge S)$$
$$P \sqsubseteq Q \Rightarrow (S \Rightarrow P) \sqsubseteq (S \Rightarrow Q)$$

In contrast the following are also theorems:

$$P \sqsubseteq Q \Rightarrow (\neg P) \sqsupseteq (\neg Q)$$
$$P \sqsubseteq Q \Rightarrow (P \Rightarrow S) \sqsupseteq (Q \Rightarrow S)$$

Based on the above we can observe that

Theorem 2: *If function $f : L \to L$ is defined in a way that its argument is only used in a logically positive position (is never negated or on the left side if implication), then f is monotonic under the refinement ordering.*

Because refinement monotonicity is easy to observe it is rarely proven and often assumed in UTP theories. It is though a key property of all UTP defined operators and healthiness conditions.

In case of healthiness conditions (based on Theorem 1)it allows us to claim that as long as they are monotonic, we can assume that the theory they describe/generate is a complete lattice.

Theorem 3: *If healthiness conditions are monotonic under $\sqsubseteq$ then for a set HL of all the healthy predicates, $(HL, \sqsubseteq)$ is a complete lattice.*

Because the set of fixed points is a complete lattice we can conclude (based on Theorem 1) that it has both a top and a bottom element and so a least fixed point. The guaranteed existence of the least fixed point has been used in UTP to define recursion. That way a recursive function,

$$F =_{def} \lambda X \bullet F(...X...X...)$$

in UTP is defined as a least fixed point of F - μF. In order to simplify the notation we overload the definition of μ by skipping lambda character

$$\mu X \bullet ...X...X... =_{def} \mu \lambda X \bullet ...X...X...$$

The need for the *ok* observational variable

In the UTP book *ok* for designs is defined as:

ok — **records the observation that the program has been started.**

ok′ — **records the observation that the program has terminated. Here termination means proper normal termination, without error messages etc.**

[HH98]p.75

While in case of designs this explanation of *ok* makes sense it is often confusing in reactive theories where the *wait* variable can be observed. While *wait* plays a very different role than *ok*, it is defined as a boolean variable recording if the program has been started (*wait*) and terminated (*wait'*). For that reason in reactive theories we prefer to describe *ok* as a variable talking about stability of the program rather than its start and termination even though its role in the theory is the same (see section 3.3 for details).

To understand the need for *ok* we have to consider an example of a non-terminating loop ($While(true)\{\}$) and its behaviour in a theory where *ok* can not be observed. In UTP live-lock can be defined as a weakest fixed point of identity function.

$$\mu X \bullet X$$

Because any predicate is a fixed point of identity function, it is equivalent to $true$. If we now consider that the live-lock is followed by an assignment we will get:

$$\begin{array}{l} \quad \mu X \bullet X; x := 1 \\ \equiv \\ \quad true; x := 1 \\ \Downarrow \\ \quad x' = 1 \end{array}$$

Live-lock is a program that never terminates, and yet theories devoid of *ok* allow it to be sequentially followed by a correct program that will be performed (in the above case assignment will constrain the final value of x). In the theory of designs (where *ok* is observed) we ensure that every process that starts in an unstable state will diverge and will have unpredictable behaviour $P = P \vee \neg ok$, that way:

$$\begin{array}{l} \quad \mu X \bullet X; x := 1 \\ \equiv \\ \quad true; x := 1 \\ \equiv \\ \quad true; \neg ok \vee x := 1 \\ \equiv \\ \quad (true; \neg ok) \vee (true; x := 1) \\ \equiv \\ \quad true \vee (true; x := 1) \\ \equiv \\ \quad true \end{array}$$

In other words, with the help of *ok*, once a process becomes unstable, all the following processes become unstable as well.

2.2 Priority

Priority is what appears a very basic and intuitively simple concept, used to express that one thing(in this work it is an event or termination) is regarded as more important than another. In process algebras priority usually appears as a special case of a choice operator or parallelism. The prioritised parallel construct is used when we have parallel processes that must be scheduled to run sequentially, and we want the scheduler to favour one over the other. In other words, it only makes sense if parallelism is implemented as sequential interleaving.

This work focuses only on prioritised choice over process events. It also presents a strong belief that priority should only be introduced into timed theories. In its view lower priority choice can only be chosen when all the higher priority processes offer no observable behaviour within a specified time.

2.2.1 Using Priority

The notion of priority is very rare in programming languages. The reason is that a preference can be easily expressed using the `if` statement.

```
if ( priority(A)>priority(B) ) then A else B
```

Unfortunately excessive use of state variables can be expensive to reason about. Also in the case of process algebras, algebraic laws of the language can not be used without an analysis of state which undermines the reasons to use process algebras in the first place.

The most common usage of the idea of priority or preference are interrupts and scheduling algorithms (Chapter 7). Priority can be used to remove nondeterministic behaviour [Low93], undesired in hardware. Finally priority translates into simple hardware logic [BW05a]. Because of that, a notion of priority has found its way into languages like occam [Bar89] or Handel-C [Cel05].

In the case of CSP, priority allows us to break symmetry of all the existing operators. With asymmetric operators we can treat two processes or events differently, something that in CSP could only be possible using the `if` statement. An algebraically asymmetric operator allows us to reason about special kinds of communication like broadcasting or asynchronous communication, where successful communication has to have priority over the failure. All the above reasons to talk about priority assume the notion of time and because of that a deadline for the prioritised choice to be determined. For that reason this work focuses on the timed flavour of *Circus*: slotted-*Circus*.

Worth noticing though is an interesting aspect of priority in untimed theories which is the specification of timed theories. The most popular way of describing timed systems using CSP is by introducing an event *tick* along with some kind of clock process $CLOCK = tick \rightarrow CLOCK$. Unfortunately this solution has always one disadvantage - *tick* in CSP is an ordinary event and *CLOCK* is an ordinary process and so it is impossible to guarantee the maximum urgency of events. In order to do that we would have to make sure that *CLOCK* is not *tick*ing without any regard to the possibility of communication and this can only be achieved by breaking the symmetry of CSP and stating that the *tick* event has always lower priority than any other possible events.

2.2.2 Priority in Existing Theories

Priority in occam

occam is a programming language based on the concepts of EPL (Experimental Programming Language) [MWS78] and CSP (Communicating Sequential Processes) [Hoa85a]. Its prime concepts are concurrency and communication, and its key targets are description of parallel algorithms along with their implementation [INM88]. occam introduces a notion of priority for both *ALT* (analogous to CSP's external choice) and *PAR* (parallel construct). Unfortunately, the occam manual does not precisely say how its prioritised operators work, or what their limitations are.

The lack of focus on this aspect of occam resulted in a number of papers about problems with priority, including with the most well known implementation of occam language - the Transputer.

"The most significant failing of the transputer implementation of PRI ALT is that it does not have a simple semantics, indeed that the easiest way of explaining what it means is to describe what it does." [BGJK88b]

Unfortunately those papers focus on fixing the implementation of priority rather than on the mathematical theory behind it. Because of that there are still cases where priority might not work as expected. [BGJK88a].

The reason for those problems, as later explained, is a strong connection between the priority and time. Especially important here is a precise notion of deadline or time bounds for a choice to be resolved.

Priority in CCS

The introduction of priority in occam language resulted in a very strong interest in this topic in the CCS research area. The research performed can be divided in to two groups: occam-based (presented in the previous section) and the theoretical work that tries to define priority with strong and intuitive algebraic properties. The scope of that work can be seen in [LNN99] in which different solutions are being classified depending on their purpose and the way priority is expressed. A good example of prioritised CCS is defined in [CW95]. Especially interesting are the priority laws presented there and the fact that they overlap with some of the results achieved in this work. Unfortunately the differences in the way communication is handled in CCS is limiting the reusability of the solutions presented there. One of the best examples of the difference between prioritised choice in CSP and prisum in CCS is that in CCS hiding commutes over the choice operator and so it does over prisum, while it is not the case in CSP.

$$(A+B)\backslash L =_{CCS} A\backslash L + B\backslash L$$

but,

$$(a \rightarrow P \,\Box\, Stop)\backslash\{a\} =_{CSP} P\backslash\{a\}$$

$$(a \rightarrow P)\backslash\{a\} \,\Box\, Stop\backslash\{a\} =_{CSP} P\backslash\{a\} \,\Box\, Stop$$

That difference has even stronger consequences in the timed theory where race conditions are a major aspect of external and prioritised choice.

Priority in Mathematics and Economics

In mathematics most of the work on preference is focused on the voting problem where people create an ordered list of their preferences. Unfortunately the notion of priority presented in this work gives us no statistical tools needed to count or weight the individual votes. In other words if we have a hundred processes preferring a above b synchronised with a single process that prefers the opposite, then our notion of priority will decide that both a and b have the same priority. For the same reason the presented notion of priority does not fit preference theories defined in economy theory. An exception is a mathematical discussion concerning the voting paradox(Condorcet's paradox) [Con85], intransitivity and cycles that can occur with preferences [BHM88].

"The proof of intransitivity is a simple example of reductio ad absurdum. If the individual is alleged to prefer A to B, B to C, and C to A, we can enquire which he would prefer from the collection of A, B, and C. Ex-hypothesi he must prefer one, say he prefers A to B or C. This however contradicts the statement that he prefers C to A, and hence the alleged intransitivity must be false."[Ana93]

2.3 Handel-C tutorial

In a simplified view Handel-C has a C syntax with an addition of non standard operators for handling parallelism, communication and hardware specific issues like memory or clock. In this section we will briefly introduce the core operators with a focus on communication, parallelism and preference. We will also introduce the undergoing details of timing constrains that are important in understanding similarities between Handel-C and slotted-*Circus*.

2.3.1 Operators

The best way to introduce the Handel-C constructs is to compare them to the ones available in ANSI-C.

ANSI-C and Handel-C	Handel-C only
{ ; }	par
switch	delay
do ... while	?
while	!
if ... else	prialt
for (;;)	seq
break	ifselect
continue	
return	
goto	
assert	

[Cel05]

All the ANSI-C operators have the standard meaning, which makes the Handel-C code easy to understand. In order to implement a time efficient hardware, where possible, we need to use parallelism. To do that Handel-C provides `par` operator.

```
x=2;
par{
    x=0;
    y=x;
}
z=x+y; // z==2
```

The parallel operator allows to list commands that will be executed at the same clock cycle. Because variables are updated at the end o a clock cycle and after all the expressions are evaluated, in the example above $y = x$ will assign to y the value of $x = 2$.

For communication within the Handel-C program we use syntax similar to CSP.

```
Channel ?  Variable; //reads from a channel
Channel !  Expression; //writes to a channel
```

Worth noting is that the use of two types of channels are possible. The buffered one allowing non blocking communication that will store the data in a buffer until it is collected, and the unbuffered type, similar to solutions used in the CCS family of languages, where the read and write processes will wait until communication between them is possible. Only two processes can take part in a communication (point to point). Because the buffered type can be implemented using the unbuffered communication we will be more interested with the second.

2.3.2 Timing details

The basic rule of timing constrains in Handel-C is:

Assignment and delay take 1 clock cycle. Everything else is free.

[Cel05]p.169.

In consequence of this rule a communication (both send and receive) also takes a clock cycle because the received data needs to be recorded in memory.[3] Also the above rule implies that the complexity of expressions

[3]The timing property of communication can be walked around by the use of signal variables and the "prialt trick" that is presented as one of the case studies (section 7.1).

that are evaluated before they are assigned is not important, and so expressions can always be evaluated within a clock cycle. While that is true, the complexity of hardware logic that is designed to be run within a clock cycle, has a direct influence on the clock rate with which the hardware will be able to run.

2.3.3 Signal variables

An interesting aspect of Handel-C, that allows immediate exchange of information, is the signal variable. Signal variable is in fact a type of communication where channel has a form of a variable. Applying a value to a signal variable sends the value, while reading the variable collects it. Because the value of a signal variable is not stored, it takes no time for assignment and write it. The obvious problem is that the value of a signal variable is only accessible during the clock cycle when it was assigned. Also a signal variable allows description of un-implementable logic loops and enforces compiler to perform extra static checks on the code.

```
int 15 a, b;
signal <int> sig;
a = 7;
par {
    sig = a;
    b = sig;
}
```

[Cel05]p.89.

In the example above b will be equivalent the value of $a = 7$.

The signal variable can have a default value that will always be used when no value was assigned during the clock cycle.

```
int 15 a, b;
static signal <int> sig = 100;
a = 7;
par {
    sig = a;
    b = sig;
}
a = sig;
```

[Cel05]p.90.

In here the final value of a will be the default of $sig = 100$ and the value of b will be 7. This is because the assignment $sig = a$ is one clock cycle earlier than $a = sig$ ($b = sig$ takes one clock cycle and is delaying the whole $par\{...\}$ block).

2.3.4 `prialt`

prialt operator has a syntax similar to `switch`, but instead of logical tests it has a list of possible communications. Where with `switch` the available options are executed when the guarding logical test is true, with `prialt` communication needs to be performed. If few communication options are possible then the top priority option (highest on the list) is chosen.

```
prialt
{
        case a ?  x:
             x++;
             break;
```

```
        case b !  y:
             y++;
             break;
}
...
```

The presented code awaits for input on channel `a` and output on channel `b`. If none of the communications are available then the prialt construct will allow time to pass. If only one case is available within a clock cycle then `prialt` will execute it. If both of the cases are available within the same clock cycle then `prialt` will execute the higher priority option (`a ?  x:  x++;`) which will read a value on the input channel to the variable, perform a clock tick (communication takes time), increment the variable, perform a clock tick (assignment takes time) and terminate.

`prialt` statement can also be used with the `default` case.

```
prialt
{
        case a ?  x:
             x++;
             break;
        case b !  y:
             y++;
             break;
        default:
             z++;
             break;
}
...
```

While without `default` the execution of `prialt` halts until one of the cases is ready to communicate, the presence of `default` means that if no communication is available immediately then the `default` branch will be performed instead. It is important to note here that the default branch will be performed without any delays, opposite to the standard cases which always have to reserve one clock cycle for communication. The detailed description of the semantics of `prialt` can be found in [BW02].

Chapter 3

slotted-*Circus*

3.1 Contribution

The first definitions and syntax of the slotted-*Circus* actions and operators were based on the work of Adnan Sherif and He Jifeng on Circus Timed Actions (CTA) [SH03], fully presented in [She06]. Sherif's work mostly originated from *Circus*, which semantics had its roots in the UTP semantics of CSP. The idea of slotted-*Circus* was first described by Andrew Butterfield [BSW07]. It introduced the exchangeable history models along with a precise, layered, low level semantics of slotted-*Circus* (trace, slot, slots and actions layers). The top level of slotted-*Circus* (definitions of actions and operators) was a syntactically adapted version of CTA. Later work on slotted-*Circus* has been performed with the help and supervision of Andrew Butterfield, and is presented in this thesis. As the theory has moved from its CTA origins various aspects of the definitions have been gradually improved, correcting errors and improving the useability. Operators affected by this research include Skip, assignment, external choice, prefixing, parallelism and hiding. Of particular interest was a feature of the semantics that went all the way back to *Circus* itself in which having the variable state visible during waiting could lead to a miraculous behaviour.

The state visibility problem was addressed in cooperation with Jim Woodcock from the University of York and resulted in [BGW09]. The changes to the theory, while being small, had to be made to one of the healthiness conditions and therefore they had to be applied to most of the defined actions and operators.

For the reasons explained above, the definitions of lower levels of slotted-*Circus* have been placed in the appendix for the purpose of keeping this work complete. On the other hand, the definitions of actions and operators have been included in the main body of this work due to the number of major changes that had to be introduced as well as the time and effort invested in them.

3.1.1 Corrections of CTA [She06]

While the definitions of the operators presented in this work originated from CTA, many errors have been found and needed to be addressed in [She06]. The operators that proved to be problematic are listed below, together with the explanations of the changes that needed to be applied.

$$[\text{Skip:def:CTA}] \qquad Skip_{CTA} \mathrel{\widehat{=}} \mathbf{R}_t(\exists ref \bullet snd(last(tr_t)) \wedge \mathbb{I}_t)$$

(p.39, (3.5.2), [She06])

The problem in here is that the above definition can be reduced to

$$Skip_{CTA} = \mathbf{R}_t(\mathbb{I}_t)) \qquad ,$$

which means that $Skip_CTA$ does not leave the refusals set (of the last slot) unconstrained. This error has

a negative influence on the usefulness of $CSP3$ and $CSP4$ healthiness conditions, resulting in an incorrect behaviour of the hiding operator. A similar issue applies to the definition of assignment (p.39, (3.5.7), [She06]).

The opposite problem can be observed in the use of $trace'_{CTA}$ short hand notation.

$$[\text{trace':def:CTA}] \qquad trace'_{CTA} \mathrel{\widehat=} Flat_{CTA}(tr') \frown Flat_{CTA}(tr)$$
$$trace' : seqEvent$$

(p.32, (3.3.1), [She06])

$trace'_{CTA}$ extracts a sequence of events that occurred between the starting (tr) and final (tr') trace. In other words, it removes all the information about time and refusals from the recorded trace ("flattens" it) and shows the difference between the before and after-trace. Because $trace'_{CTA}$ by itself is not prefix closed, it results in major issues in the definition of $Stop_{CTA}$, making it not prefix-healthy.

$$[\text{trace':def:CTA}] \qquad Stop_{CTA} = CSP1_t(\mathbf{R}3_t(ok' \wedge wait' \wedge trace' = \langle\rangle))$$

(p.39, (3.5.5), [She06])

In many other definitions this problem is, somehow accidently, fixed by an application of $\mathbf{R}2$ healthiness. Unfortunately it is not the case with the prefix definition, where uncareful treatment of refusals sets negates the authors attempts to ensure the maximum urgency rule (events should be performed as soon as possible).

In the case of external choice in CTA we deal with the state visibility issue which is explained in detail in section 3.6.9. Also major effort has been invested into the simplification of the definition with regard to the complexity of the proofs of major properties.

Finally a problem of synchronisation of parallel processes has been addressed by a condition, enforcing that clocks of parallel actions can be de-synchronised only when one of them terminates.

$$(wait_A \Rightarrow \#slots_A \geq \#slots_B) \wedge$$

$$(wait_B \Rightarrow \#slots_B \geq \#slots_A)$$

Detailed explanation of this case can be found in the subsection 3.9.7.

3.2 slotted-*Circus* Syntax

The slotted-*Circus* syntax is very similar to that of *Circus*. There are two classes of primitives: events and actions (processes in CSP). Events represents communication or interaction of actions. Actions represents behaviour or systems that can perform events or combine behaviour of other actions.

$$
\begin{array}{rcl}
\mathsf{Action} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \\
& \mid & \mathsf{Name}^+ := \mathsf{Expr}^+ \mid \mathsf{Comm} \rightarrow \mathsf{Action} \\
& \mid & \mathsf{Action}; \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \\
& \mid & \mathsf{Action} \lhd \mathsf{b} \rhd \mathsf{Action} \mid \mathsf{Action} \setminus \mathsf{CS} \mid \mu \mathsf{Name} \bullet F(\mathsf{Name}) \\
& \mid & \mathsf{Action} \underset{\mathsf{CS}}{\parallel} \mathsf{Action} \mid \mathsf{Action} \parallel \mathsf{Action} \mid \mathsf{Action} \mid\mid\mid \mathsf{Action} \mid \mathit{Wait}\ \mathsf{t} \\
\mathsf{Comm} & ::= & \mathsf{Name} \mid \mathsf{Name.Expr} \mid \mathsf{Name!Expr} \mid \mathsf{Name?Name} \mid \mathsf{Name} : \mathsf{T} \\
\mathsf{CS} & ::= & \{ \mathsf{Comm}^+ \} \mid \{\mid \mathsf{Comm}^+ \mid\} \\
\mathsf{T} & ::= & \text{type} \\
\mathsf{Expr} & ::= & \text{expression} \\
\mathsf{t} & ::= & \text{positive integer valued expression} \\
\mathsf{b} & ::= & \text{boolean valued expression} \\
\mathsf{Name} & ::= & \text{channel or variable names} \\
\mathsf{CS} & ::= & \text{channel name sets}
\end{array}
$$

There are three special processes *Skip*, *Stop* and *Chaos* where *Skip* represents the process that does nothing and terminates, *Stop* is a deadlocked action that does not perform any events and does not terminate and *Chaos* is an action that is unpredictable and can do anything.

List of available operators is short and very similar to CSP.

The **prefix** operator creates a process from an event (a) and another process (P).

Action S equal to:

$$a \rightarrow P$$

means that S will communicate (perform an event) a with the environment (if the environment is willing to do that) and then it will behave like action P. If the environment is not ready to perform the event then the action will patiently wait.

$$\mathit{CoffeeMachine} = \mathit{getMoney} \rightarrow \mathit{makeCoffee} \rightarrow \mathit{CoffeeMachine}$$

For example a coffee machine first get some money that it makes coffee, but in both of the cases it cooperates with the environment (it can not get money without anyone putting a coin in it, and it can not finish the coffee making process without anyone collecting it).

The **nondeterministic choice** operator, often called internal choice, creates an action from two actions (P, Q). Action S equal to:

$$P \sqcap Q$$

means that S will behave like either P or Q.

$$\mathit{RandomBits} = 1 \rightarrow \mathit{RandomBits} \sqcap 0 \rightarrow \mathit{RandomBits}$$

RandomBits action nondeterministically performs 1 or 0 and behaves like *RandomBits* again.

The **deterministic choice** operator, often called external choice, creates an action from two other actions (P, Q). Action S equal to:

$$P \Box Q$$

means that, depending on the environment, S will behave like either P or Q. The deterministic choice is resolved by the environment when it agrees on the first event that either P or Q is offering. If the environment does not

agree to perform the first event from P and Q then the deterministic choice is equal to $Stop$

$$FullPlate = eatLunch \rightarrow EmptyPlate \Box wait \rightarrow FullPlate$$

Depending on the environment $FullPlate$ remains full or gets empty if someone will eat the lunch.

The **parallel** operator is available in a few options. We can supply a set of events that two processes will synchronise on ($A \underset{CS}{||} B$), or we can use the fully synchronised parallel ($A \,||\, B$), or finally we can use the fully asynchronous parallel (interleaving) ($A \,|||\, B$).

$$Lunch = FullPlate \| HungryMan$$

$$HungryMan = eatLunch \rightarrow HappyMan \Box wait \rightarrow HungryMan$$

$HungryMan$ is willing to synchronise with $FullPlate$ either on waiting or eating the lunch.

$$Network = Server \,||\, (Client1 \,|||\, Client2 \,|||\, Client3)$$

In here we present a network of a few clients and a single server. Clients are only allowed direct contact with the server.

Sequential Composition is used when two processes are run one after another. Like in many sequential programming languages it is denoted by a semicolon.

$$TWODAYS = DAY; DAY$$

Assignment is an action that assigns values to variable

$$INCX = x := x + 1$$

To simplify the theory and the syntax we currently do not define complicated types (in fact we don't mention types at all). In slotted-*Circus* we only talk about variables and expressions leaving the details open to implementors.

The **conditional** is used for the basic boolean tests. In slotted-*Circus* $A \lhd b \rhd B$ is read as "If b then A else B".

$$ABSX = (x := -x) \lhd x < 0 \rhd Skip$$

In slotted-*Circus* **Recursion** is denoted as the weakest fixed point with respect to the refinement lattice ordering (subsection 2.1.4).

$$WHILELOOP = \mu P \bullet (LOOPBODY; P) \lhd cond \rhd Skip$$

Hiding encapsulates the scope of events. Events that are hidden become internal and can no longer be observable. Also, events that are hidden are performed immediately and no longer have to wait for the environment to agree on them.

$$PingPong = \left(\begin{array}{l} (ping \rightarrow synch \rightarrow Skip) \\ || \; (synch \rightarrow pong \rightarrow Skip) \end{array} \right) \setminus \{ synch \} = ping \rightarrow pong \rightarrow Skip$$

When we use hiding or parallel construct on channels that are passing some data, we can use a special notation $\{| \, channelName \, |\}$ which will generate a set of all possible events of $channelName.x$. Currently there is no need to define the type of x.

$$TwoBitBuff = \left(\begin{array}{l} (\mu BitBuff \bullet in.x \rightarrow mid.x \rightarrow BitBuff) \\ \underset{\{|mid|\}}{||} \\ (\mu BitBuff \bullet mid.x \rightarrow out.x \rightarrow BitBuff) \end{array} \right) \setminus \{| \, mid \, |\}$$

Finally the **wait** operator is used to describe a process being idle for a given time.

$$TimeoutEvent = (event \rightarrow Skip) \Box (Wait \; t)$$

In here if *Wait* will terminate after time *t* and if *event* still wasn't performed then the whole action will terminate.

Introduction of time along with the *Wait* operator allowed us to implement a racing condition into external choice.

$$Race = (start \rightarrow win \rightarrow lose \rightarrow lose \rightarrow Skip) \; || \; (Rabbit \underset{\{start\}}{||} Turtle \underset{\{start\}}{||} Donkey)$$

We describe a race of a rabbit, donkey and a turtle. The race is synchronised with players on a single *start* event (one start for all three players who share this event) and separate *win*, *lose* and *lose* events for each of the players.

$$Rabbit = start \rightarrow Wait \; 4; (win \rightarrow Skip \; \Box \; lose \rightarrow Skip)$$

$$Turtle = start \rightarrow Wait \; 7; (win \rightarrow Skip \; \Box \; lose \rightarrow Skip)$$

$$Donkey = start \rightarrow Wait \; 5; (win \rightarrow Skip \; \Box \; lose \rightarrow Skip)$$

All the players perform start event then introduce a delay using *Wait* operator. After the desired time will pass they win or lose, depending on the environment. Because of the racing condition implemented in external choice *Rabbit*, will choose the first available option - *win* (assuming that environment will not interfere). Similar other players, given choice between waiting (in a hope that if they wait long enough they might be able to win) or losing, will choose to lose.

It is possible in this example that the environment will block some of the events from occuring, cousing the events of the race to be delayed or never performed.

$$NuclearBomb = start \rightarrow boom \rightarrow Stop$$

$$(NuclearBomb \underset{\{start,win,loose\}}{||} Race) \setminus \{boom\} = start \rightarrow Stop$$

We can address that issue by seperating the internal events of the race from the environment

$$Race = \left(\begin{array}{l} (start \rightarrow win \rightarrow lose \rightarrow lose \rightarrow Skip) \\ || \\ (Rabbit \underset{\{start\}}{||} Turtle \underset{\{start\}}{||} Donkey) \end{array} \right) \setminus \{start, win, loose\}$$

3.3 Alphabet

As mentioned before, every UTP based semantics is defined as a set of predicates over an alphabet. The alphabet consists of two sets of variables "undashed" (*obs*) and corresponding "dashed" (obs') variables.

$$\mathbb{A} = obs \cup obs'$$

In slotted-*Circus* the alphabet $\mathbb{A}$ consists of four couples of variables ($slots$, $slots'$, ok, ok', $wait$, $wait'$, $state$, $state'$).

$$obs = \{slots, ok, wait, state\}$$

$$obs' = \{slots', ok', wait', state'\}$$

$slots$ and $slots'$ represent the recorded trace including information about refusals sets and time. The details of how these information are recorded can be found in section 3.5.

The variables representing state ($state$, $state'$) are left undefined and only basic tools are provided to access and edit them - state update operator $\oplus$ and expression evaluator $val(e, state)$(the syntax or type of possible expressions is undefined) .

$$state' = state \oplus \{x \mapsto val(e, state)\}$$

(Update variable x within $state'$ with the value of the expression e see subsection 3.8.6)

The last two pairs of observation variables are used to control reactions of an action on other processes that are sequentially composed with it. $wait$ and $wait'$ records wether an action has started or finished, while ok and ok' stores information about its stability (for details see 2.1.4). Below we present possible values of the four variables and their meaning.

- $(\neg ok)$ started in an unstable state.

- $(ok \wedge \neg wait)$ a process that diverged during execution (note that ok' is unconstrained see [CSP2:def]:p. 27 for details).

-$(ok \wedge wait \wedge ok' \wedge wait')$ stable and waiting to start.

-$(ok \wedge ok' \wedge \neg wait \wedge wait')$ started, is stable, but did not terminate.

-$(ok \wedge ok' \wedge \neg wait \wedge \neg wait')$ started and terminated in a stable state.

3.4 Layered Model of the Denotational Semantics

In slotted-*Circus*, the UTP semantics is built using several levels of abstraction. Each has its own definitions of operators like addition or subtraction, different notions of equivalence and ordering, along with a number of algebraic properties which are often used in proofs.

The single $slot$ level represents a history of actions during one time unit and is defined as a couple $(hist, ref)$. The history of events that happened during a clock cycle is recorded using $hist$ while ref informs us which events are being refused in the time unit. There are currently two history models (defining $hist$ part of a $slot$) available for slotted-*Circus* - *CTA* and *MSA*, which we discuss later.

Using $Slot$ and its operators we can defined $slots$ that is a nonempty sequence of $slot$'s.

$$slots : slot^+$$

$slots$ level works as a trace for the denotational semantics of slotted-*Circus* and most of it's operators have inductive definition over $slot$ and $slot$-level operators.

Finally, using $slots$, we can define some very basic behaviours like $NOEVENTS$, or $EVTSNOW$ that create the last layer of abstraction, needed to define the language operators and healthiness conditions.

3.5 History Models

The idea of generic history type allows us to define and use different history models without redefining slotted-*Circus* and being forced to redo the proofs of its laws. Every instantiations of history in slotted-*Circus* has to follow a number of properties and implement a list of operators which are used on the higher levels of slotted-*Circus* semantics.

One of the theories that can be instantiated in slotted-*Circus* is *CTA* [She06]. In *CTA* the history model is build of slots containing sequences of events. Because events during one time unit (slot) are remembered as a sequence, their ordering is important.

$$slotHist_{CTA} \in Event^*$$

In CSP, in order to distinguish external(deterministic) choice from internal(nondeterministic) we need extra information about which events are currently being refused by the process. In timed theories one refusal set per slot sequence is not enough. In order to properly define external choice it is necessary that refusals are recorded for every time unit. For that reason the type of the *CTA* slot is:

$$slot_{CTA} \in Event^* \times \mathbb{P}E$$

An example of *CTA slots* trace could be:

$$\langle(\langle\rangle, ref_1), (\langle a,b,a\rangle, ref_2), (\langle c,a,a\rangle, ref_3)\rangle$$

In here we describe a situation when a process does nothing for one clock cycle, sequentially performs a, b and $a$during the second clock cycle, and finally performs c,a and a in the third clock cycle. Clock ticks always occur between clock cycles, and there is no need to implicitly record them in the trace model.

The second existing history model is *MSA* where we only count the number of occurrences of different events without mentioning their ordering. We model these using multisets (or bags).

$$slotHist_{MSA} \in E \to \mathbb{N}$$

$$slot_{MSA} \in (E \to \mathbb{N}) \times \mathbb{P}E$$

An example of *MSA slots* trace corresponding but not the same as one presented above could be:

$$\langle(\{\,\}, ref_1), (\{a \mapsto 2, b \mapsto 1\}, ref_2), (\{a \mapsto 2, c \mapsto 1\}, ref_3)\rangle$$

As we can see, while the information about the occurrence of events is in the presented *CTA* and *MSA* traces is the same. The *MSA* trace loses the information about the order in which they were performed. The consequences of this are discussed in detail throughout the chapter 4

Once the trace model is defined we can use operators like ordering, equality, addition and others to define the UTP semantics of slotted-*Circus*.

3.6 Healthiness Conditions

The healthiness conditions in slotted-*Circus* are analogues to the reactive and CSP healthiness conditions of UTP, namely **R** (**R**1,**R**2,**R**3) and CSP1–5. The adaptation of the healthiness conditions to the timed theory has been done by Adnan Sherif [She06], followed by the work of Andrew Butterfield [BSW07]. In this section we will focus on interesting properties, errors and observations that has been explored during this research. The full list of important properties can be found in the appendix (A.4).

3.6.1 Reactive Healthiness 1 (R1)

The first of the healthiness conditions ensures that a healthy action can only add events to its history. This protects us from actions that would try to change the past (edit the history). In *Circus* a **R**1 healthy process asserts that $tr \leq tr'$. In a slotted theory, we use the slots ordering operator $\preccurlyeq$. While the meaning of $\preccurlyeq$ is obvious its exact definition depends on the history model used. $slots \preccurlyeq slots'$ states that $slots$ is a prefix of $slots'$ for all but a last slot. The history at last slot of $slots$ is at the same time a "precursor" (the meaning depends on the trace model) of the the corresponding (the same in order) slot history in $slots'$.

$$[\text{GROW:def}] \qquad GROW \mathrel{\widehat=} slots \preccurlyeq slots'$$

$$[\text{R1:def}] \qquad \mathbf{R}1(P) \mathrel{\widehat=} P \wedge GROW$$

3.6.2 Reactive Healthiness 2 (R2)

R2 healthiness enures that the history of the previous actions has no influence on the behaviour of the current one. In other words, an **R**2 healthy actions behaves the same for any recorded history - the set of accepted changes to traces ($slots' \nwarrow slots$) is the same, no mater what was the initial value of the recorded trace ($slots$). In result the only information about the starting state of an action (not counting ok and $wait$ which play a technical role) can be recorded in the $state$ variable. We ensure that the value of $slots$ observation has no influence on the behaviour of an action.

$$[\text{R2:def}] \qquad \mathbf{R}2(P) \mathrel{\widehat=} \exists ss \bullet P[ss, ss \# (slots' \nwarrow slots)/slots, slots'] \wedge eqvref(ss) = eqvref(slots)$$

Where $\#$ is a slots addition operator [CAT:def]:p. 109, $\nwarrow$ is a slots subtraction operator [DF:binop]:p. 110, and $eqvref(ss)$ is a function extracting the last refusals set from slots [ER:def]:p. 107.

See [But11b] and [But06] for details of how this formulation was developed.

3.6.3 Reactive Healthiness 3 (R3)

R3 encapsulates the principle that while a healthy process is waiting to be started it does nothing and behaves like identity. That way we ensure that for a healthy actions P and Q, if P is not terminated then Q is a unit of sequential composition:

$$(P \wedge wait'); Q = (P \wedge wait'); Skip$$

An important aspect of the definition of **R**3 is that it is using reactive identity and so it is not copying the $state$ variables of the previous process. The reasons for that are explained in the section 3.6.9.

$$[\text{DIV:def}] \qquad DIV \mathrel{\widehat=} \neg ok \wedge GROW$$

$$[\text{IIr:def}] \qquad \mathit{I\!I}_R \mathrel{\widehat=} DIV \vee ok' \wedge wait' \wedge slots' = slots$$

$$[\text{R3:def}] \qquad \mathbf{R}3(P) \mathrel{\widehat=} \mathit{I\!I}_R \lhd wait \rhd P$$

$$[\text{R3:idem}] \qquad \mathbf{R}3 \circ \mathbf{R}3 = \mathbf{R}3$$

3.6.4 Reactive Healthiness (R)

We often use a shorthand for the composition of all the reactive healthiness conditions.

$$[\text{R:def}] \qquad \mathbf{R} = \mathbf{R}3 \circ \mathbf{R}2 \circ \mathbf{R}1$$

The order in which **R**1 , **R**2 and **R**3 are composed is irrelevant because all the healthiness conditions commute.

3.6.5 CSP Healthiness 1 (CSP1)

A process is CSP1 -healthy if it behaves like DIV when started in an unstable state. This property is important for the proper behaviour of recursion and is fully explained in the section 2.1.4.

$$[\text{CSP1:def}] \qquad CSP1(P) \mathrel{\widehat=} P \vee DIV$$

3.6.6 CSP Healthiness 2 (CSP2)

CSP2 is another healthiness condition that ensures that recursion is well behaved. It supports CSP1 by ensuring that any divergent action is CSP4 healthy. Also it standardises the way divergence is signaled (instead of stating $\neg ok'$, we always DO NOT state that the divergent action is ok', by leaving ok' unconstrained).

$$[\text{CSP2:def}] \qquad CSP2(P) \mathrel{\widehat{=}} P; ((ok \Rightarrow ok') \wedge wait' = wait \wedge state' = state \wedge slots' = slots)$$

$$[\text{CSP:alt}] \qquad \exists ok_0 \bullet P[ok_0/ok'] \wedge (ok_0 \Rightarrow ok')$$

3.6.7 CSP Healthiness 3 (CSP3)

CSP3 simply states that *Skip* is a left unit of sequential composition. It is never used as a part of definition and is not supported by all of the operators (hiding is not CSP3 healthy). For that reason it might seem like CSP3 is not necessary in the theory and should be only noted as one of the laws. The truth is that CSP3 needs to be viewed as a twin property to CSP4 . Because CSP4 is often used in the definitions it is ensured that every operator is CSP4 healthy, and we observe that CSP4 actions are always "healthifying" any following, CSP3 unhealthy actions.

$$\begin{aligned} & CSP4(P); Q \\ = \; & (CSP4(P); Skip); Q \\ = \; & CSP4(P); (Skip; Q) \\ = \; & CSP4(P); CSP3(Q) \end{aligned}$$

CSP3 plays a key role in the correct behaviour of hiding. It unconstrains the refusal set and ensures that hiding is not restricting possible behaviours based on the refusals of a previously terminated action.

$$[\text{CSP3:def}] \qquad CSP3(P) \mathrel{\widehat{=}} Skip; P$$

The definition of *Skip* is given later.

3.6.8 CSP Healthiness 4 (CSP4)

As explained before CSP4 is responsible for unconstraining the refusals of a terminated process. It is also responsible for unconstraining the state of a process that has not terminated. This property is explained in detail in the following subsection 3.6.9.

$$[\text{CSP4:def}] \qquad CSP4(P) \mathrel{\widehat{=}} P; Skip$$

3.6.9 Fixing the State Visibility

The state visibility problem in slotted-*Circus*, *CTA* and *Circus* was detected while trying to prove some of the state-based properties of external choice. The problem originated from the standard definition of the choice operator in the UTP semantics of CSP:

$$A \Box B =_{CSP} CSP2(A \wedge B \lhd Stop \rhd A \vee B)$$

In the semantics of *Circus*, *CTA* and slotted-*Circus* the definition of $\neg Stop \wedge (A \vee B)$ has been changed. Unfortunately the behaviour of external choice before the decision is made has been left unchanged as $Stop \wedge A \wedge B$.

The problem here was that with the introduction of state both A and B could introduce changes to state without performing any communication. That resulted in an undesired behaviour of the theory.

$$
\begin{array}{ll}
 & (x := 1; Stop) \,\Box\, (x := 2; Stop) \\
\equiv & \\
 & (x := 1; Stop) \wedge (x := 2; Stop) \wedge Stop \\
\equiv & \\
 & ((x := 1 \wedge x := 2); Stop) \wedge Stop \\
\equiv & \\
 & (\mathbf{R}(CSP1(false)); Stop) \wedge Stop \\
\equiv & \\
 & (Miracle; Stop) \wedge Stop \\
\equiv & \\
 & Miracle
\end{array}
$$

While the obvious solution to this problem would be to fix the external choice operator, it was decided that it will be safer to address the deeper source of the problem, which is state visibility. A process before it terminates can perform many actions on state. Those actions are not recorded in a trace, can be performed sequentially within the same clock cycle, not separated by any events. For that reason, while a process is running, state should be considered as unstable. In *Circus* and all the theories inheriting its semantics, state does not influence its environment and so there is no need for it to be visible until the process terminates.

In order to make the behaviour of state safe and predictable it was decided to encapsulate the principle that while a healthy process is still running, its variable state is unobservable. Instead of adding a new healthiness condition to the theory we can inject the restriction on state using a combination of existing ones, namely CSP4 and **R**3 . If we take an arbitrary process P that never terminates we can trace the mechanism of how it works:

$$
\begin{array}{ll}
 & CSP4(P \wedge \neg wait') \\
\equiv & \quad \text{“ CSP4 def ”} \\
 & (P \wedge \neg wait'); Skip \\
\equiv & \quad \text{“ Skip is } \mathbf{R}3 \text{ ”} \\
 & (P \wedge \neg wait'); \mathbf{R}3(Skip) \\
\equiv & \quad \text{“ } \mathbf{R}3 \text{ def ”} \\
 & (P \wedge \neg wait'); I\!I_R
\end{array}
$$

Where, $I\!I \equiv I\!I_R \wedge state = state'$.

Here we see that **R**3 is defined using the reactive version of identity - $I\!I_R$ instead of $I\!I$, as it was done earlier. The difference is that the reactive identity does not copy the state and so information that P can have on it ($state'$) is unconstrained.

The problem and given solution was discussed and published in [BGW09]

3.6.10 Laws

A large list of properties of the healthiness conditions can be found in the appendix A.4

3.7 Basic Actions

Basic actions allow us to define behaviours that are often recurring in the definitions of slotted-*Circus* actions or create aliases for intuitively easy behaviour with a complicated definition. Basic actions also have a number of very strong algebraic properties which allow better reusability and readability of proofs. Basic actions are never healthy and they always address only the trace aspect of behaviour.

3.7.1 Notation

Because all the basic actions are predicates over $slots$ and $slots'$, we overload their notations.

$$BasicActions \mathrel{\widehat=} BasicActions(slots, slots')$$

We also use a type generator $\mathscr{S}$ for describing the type of a single *slot* ($\mathscr{S}\ E$), or $slots$ defined as a nonempty sequence of *slot* ($(\mathscr{S}\ E)^+$). $\mathscr{S}$ is a function over type E which denotes events type.

3.7.2 EQVTRACE

A common building block for all the basic actions is the $EQVTRACE$ - a $slots$-level relation allowing an extraction of sequence of events recorded in the given trace.

$$EQVTRACE : E^* \leftrightarrow (\mathscr{S}\ E)^+$$

For example:

$$EQVTRACE(nil, slots) \equiv True$$

signifies that no events have been recorded in $slots$,

$$EQVTRACE(\langle a, b\rangle, slots) \equiv True$$

means that an event a followed by an event b was recorded in $slots$.

$EQVTRACE$ ignores all the information about time and refusals. Detailed definitions and laws of $EQVTRACE$ can be found in the appendix [ETs:sig]:p. 101

3.7.3 NOEVTS

$NOEVTS$ is the most useful of the basic actions. It is analogue to $trace = trace'$ in *Circus* and describes a situation when no events were performed. The difference is that $NOEVTS$ allows time to pass (generation of new, empty slots) and changes to the last refusals set.

[NEV:sig] $$NOEVTS : (\mathscr{S}\ E)^+ \leftrightarrow (\mathscr{S}\ E)^+$$

[NEV:def] $$NOEVTS(slots, slots') \mathrel{\widehat=} EQVTRACE(nil, slots' \setminus slots)$$

3.7.4 EVTSNOW

$EVTSNOW\{e\}$ describes a situation when an event e is performed and recorded in the trace model. It is equivalent to $trace' = trace \frown \langle e\rangle$ in *Circus*. $EVTSNOW$, in contrast to $NOEVTS$, does not allow time to pass. That way it is easier to use in the prefix definition.

[EVN:sig] $$EVTSNOW : ((\mathbb{P}E) \to (\mathscr{S}\ E)^+) \leftrightarrow (\mathscr{S}\ E)^+$$

[EVN:def] $$EVTSNOW(E)(slots, slots') \mathrel{\widehat=} \exists tt \bullet elems(tt) = E \land EQVTRACE(tt, slots' \setminus slots) \land \#slots = \#slots'$$

3.7.5 IMMEVTS

Finally in some cases we would like to assert that an event has been performed in the first time slot. While in an untimed theory it can be easily described as $trace' > trace$, in slotted-*Circus* a growing trace does not guarantee that an event has occurred. What's more with $IMMEVTS$ we are interested in a very specific time bounds for the event to be performed.

[IME:sig] $IMMEVTS : (\mathscr{S}\, E)^+ \leftrightarrow (\mathscr{S}\, E)^+$

[IME:def] $IMMEVTS(slots, slots') \mathrel{\widehat=} \exists E \bullet E \neq \emptyset \land EVTSNOW(E)(slots, slots'); GROW$

$IMMEVTS$ describes an events that occurs immediately (in the first slot). Once the event is performed, it allows unconstrained expansion of the history ($GROW$). $IMMEVTS$ is used in the definition of external choice to detect that a process has performed an event. The restriction for the event to be immediate is dictated by the racing condition and the need for precise timing.

3.7.6 Laws

In here we list some of the properties of basic actions that were found useful during this research.

[BasicActionsLaw-1] $EVTSNOW(\emptyset)(ss, tt) \equiv ss \cong tt$

[proof:BasicActionsLaw-1] : $p.115$

[BasicActionsLaw-2] $EVTSNOW(\emptyset)(ss, tt) \equiv NOEVTS(ss, tt) \land \sharp ss = \sharp tt$

[BasicActionsLaw-3] $\exists s, s' \bullet EVTSNOW\{c\}(s, s') \land$
$(sl' \setminus\!\setminus sl) = map(SHide(\{c\}))(s' \setminus\!\setminus s) \equiv sl' \cong sl$

[proof:BasicActionsLaw-3] : $p.116$

Where, $ss \cong tt$ is a *slot* sequence equality operator, meaning that ss is identical to tt in all aspects but the refusals set of the last recorded slot [SSEQV:def]:p. 108. $SHide(H)(slot)$ is a slot-level operator that removes all occurrences of the events from the set H, within the history part of *slot* [SHid:def]:p. 105.

More laws of the properties of basic actiones can be found in the appendix A.5.

3.8 Actions

3.8.1 Chaos

Chaos is a healthy, stable but unpredictable action. It is also the bottom of the actions lattice, and the weakest possible specification (refined by any program).

[Chaos:def] $Chaos \mathrel{\widehat=} \mathbf{R}(\mathbf{true})$

3.8.2 Miracle

Miracle, symmetrically to *Chaos* is defined as a healthy *False*. It is a process that can never start. It is also the top of the actions lattice, strongest possible specification and most of all implementation of every specification (hence its name).

[Miracle:def] $Miracle \mathrel{\widehat=} \mathbf{R}(\neg ok)$

Here we use $\neg ok$ instead of $CSP1(FALSE)$ because:

$$
\begin{aligned}
& \mathbf{R}(CSP1(\mathbf{false})) \\
\equiv & \\
& \mathbf{R}(\mathbf{R}1(CSP1(\mathbf{false}))) \\
\equiv & \\
& \mathbf{R}((\neg ok \wedge GROW \vee \mathbf{false}) \wedge GROW) \\
\equiv & \\
& \mathbf{R}(\neg ok \wedge GROW \wedge GROW \vee \mathbf{false} \wedge GROW) \\
\equiv & \\
& \mathbf{R}(\neg ok \wedge GROW) \\
\equiv & \\
& \mathbf{R}(\mathbf{R}1(\neg ok)) \\
\equiv & \\
& \mathbf{R}(\neg ok)
\end{aligned}
$$

3.8.3 Deadlock

Stop is a stable non diverging action that performs no events and never terminates. It differs from the standard UTP definition by ensuring that while doing nothing time is allowed to pass. Because of that we can no longer state that $trace = trace'$. Instead we use previously defined $NOEVTS$ that allows generation of empty slots.

[Stop:def] $\quad Stop \mathrel{\widehat{=}} CSP1(\mathbf{R}3(ok' \wedge wait' \wedge NOEVTS))$

3.8.4 Termination

Skip is an action that does nothing but terminates. While not being necessary in the program description it is a very important part of the definitions of healthiness conditions and has very strong algebraic properties that help us to understand the behaviour of operators. *Skip* is a unit of sequential composition, is equal to *Wait* 0 ([proof:Skip:Wait0]:p. 32) and describes the boundary cases for the step laws of external choice and parallelism. An interesting aspect of this definition is that *Skip* is unconstraining refusals set of a current (last) slot. For that reason in the definition we use slots equivalence $\cong$ instead of equality. This property originates from the UTP book and is necessary to ensure *CSP*3 healthiness of *Skip* and guarantee the correct behaviour of hiding.

[Skip:def] $\quad Skip \mathrel{\widehat{=}} CSP1(\mathbf{R}3(ok' \wedge \neg wait' \wedge slots \cong slots' \wedge state = state'))$

3.8.5 Delay

The *Wait* operator is what, at the syntactic level, differentiates slotted-*Circus* from *Circus*. It describes an action that while doing nothing allows time to pass for the given amount of clock ticks. We can easily notice the similarity to the *Stop* definition. The big difference is that while *Stop* has no option of termination ($wait'$), in the definition of *Wait* we use a special condition $DELAY(t)$ that ensures the action will terminate (*DELD*) after t empty slots have been added to the trace. An interesting aspect of this definition is *state* visibility. It is an important correction explained fully in 3.6.9 and [BGW09]. We resolve the state visibility problem here, by ensuring that state is only copied once *Wait* terminates (*DELD*), and it's left unconstrained while the action is still waiting (*DELW*).

$$[\text{Wait:def}] \quad Wait\ t \mathrel{\widehat=} CSP1(\mathbf{R}(ok' \wedge DELAY(t) \wedge NOEVTS))$$

$$[\text{Del:def}] \quad DELAY(t) \mathrel{\widehat=} DELW(t) \lhd wait' \rhd DELD(t)$$

$$[\text{DELW:def}] \quad DELW(t) \mathrel{\widehat=} (\#slots' - \#slots < t)$$

$$[\text{DELD:def}] \quad DELD(t) \mathrel{\widehat=} (\#slots' - \#slots = t \wedge state' = state)$$

Below we analise a border case example $Wait\ 0$ by reducing it to $Skip$.

$$\begin{aligned}
[\text{proof:Skip:Wait0}] \quad & Wait\ 0 \\
\equiv \quad & \text{“ [Wait:def]:p. 32 ”} \\
& CSP1(\mathbf{R}(ok' \wedge DELAY(0) \wedge NOEVTS)) \\
\equiv \quad & \text{“ [Del:def]:p. 32 ”} \\
& CSP1(\mathbf{R}(ok' \wedge (DELW(0) \lhd wait' \rhd DELD(0)) \wedge NOEVTS)) \\
\equiv \quad & \text{“ [DELW:def]:p. 32 ”} \\
& CSP1(\mathbf{R}(ok' \wedge ((\#slots' - \#slots < 0) \lhd wait' \rhd DELD(0)) \wedge NOEVTS)) \\
\equiv \quad & \text{“ The process is prefix healthy ”} \\
& CSP1(\mathbf{R}(ok' \wedge (false \lhd wait' \rhd DELD(0)) \wedge NOEVTS)) \\
\equiv \quad & \text{“ Logic ”} \\
& CSP1(\mathbf{R}(ok' \wedge \neg wait' \wedge DELD(0) \wedge NOEVTS)) \\
\equiv \quad & \text{“ [DELD:def]:p. 32 ”} \\
& CSP1(\mathbf{R}(ok' \wedge \neg wait' \wedge \#slots' - \#slots = 0 \wedge state' = state \wedge NOEVTS)) \\
\equiv \quad & \text{“ Arithmetic ”} \\
& CSP1(\mathbf{R}(ok' \wedge \neg wait' \wedge \#slots' = \#slots \wedge state' = state \wedge NOEVTS)) \\
\equiv \quad & \text{“ [BasicActionsLaw-2]:p. 30, [BasicActionsLaw-1]:p. 30 ”} \\
& CSP1(\mathbf{R}(ok' \wedge \neg wait' \wedge slots \cong slots' \wedge state' = state)) \\
\equiv \quad & \text{“ [Skip:def]:p. 31 ”} \\
& Skip
\end{aligned}$$

3.8.6 Assignment

Assignment in slotted-*Circus* is defined to be almost the same as $Skip$, because it takes no observable time to work and so it terminates immediately. The only obvious difference is that assignment updates a variable stored in $state$ with the value of expression e (here val is a standard expression evaluator).

$$[\text{Asg:def}] \quad x := e \mathrel{\widehat=} CSP1(\mathbf{R}3(ok' \wedge \neg wait' \wedge slots \cong slots' \wedge state' = state \oplus \{x \mapsto val(e, state)\}))$$

In order to simplify the language we assume that the variable x is somehow recorded in state and that the expression e is correctly defined. We also do not specify what is the nature/type of the state variables. In an extreme case $state$ (and $state'$) can be assumed to be a single natural number representing computers memory and e is always a function on a single natural number ($state' = e(state)$).

3.9 Operators

3.9.1 Nondeterministic Choice

Nondeterministic choice definition follows the traditional UTP one where it is defined as disjunction.

$$[\mathsf{NDet{:}def}] \qquad P \sqcap Q \mathrel{\widehat{=}} P \vee Q$$

3.9.2 Conditional Choice

Conditional choice is similarly defined to the nondeterministic choice but the available options are guarded by a boolean conditions c and $\neg c$. Again the definition of conditional choice follows unchanged from the UTP book.

$$[\mathsf{Cond{:}def}] \qquad P \lhd c \rhd Q \mathrel{\widehat{=}} c \wedge P \vee \neg c \wedge Q$$
$$[\mathsf{Cond{:}alt}] \qquad P \lhd c \rhd Q \equiv (c \Rightarrow P) \wedge (\neg c \Rightarrow Q)$$

3.9.3 Sequential Composition

Sequential composition is defined by introducing observation variables obs_0. Similar to obs and obs', obs_0 represents $state_0, ok_0, wait_0, slots_0$ and records the middle state of the composed actions. Using obs_0 we connect the after-observations of the first action with the before-observations of the second one, basically ensuring that they are performed sequentially. Finally we hide the middle observations using existential quantification.

$$[\mathsf{Seq{:}subs}] \qquad [seq] \mathrel{\widehat{=}} [obs_0/obs] = [ok_0, wait_0, state_0, slots_0/ok, wait, state, slots]$$
$$[\mathsf{Seq{:}subs'}] \qquad [seq'] \mathrel{\widehat{=}} [obs_0/obs'] = [ok_0, wait_0, state_0, slots_0/ok', wait', state', slots']$$
$$[\mathsf{Seq{:}def}] \qquad P;Q \mathrel{\widehat{=}} \exists obs_0 \bullet P[seq'] \wedge Q[seq]$$

3.9.4 Guard

Guard is a special case of conditional choice. When the boolean expression b holds then the guard leaves an action A unchanged, otherwise it deadlocks.

$$[\mathsf{Grd{:}def}] \qquad b\&A \mathrel{\widehat{=}} A \lhd b \rhd Stop$$

3.9.5 Communication (Prefix)

The fundamental action in all the CSP or CCS based languages is prefixing (or communication). In all the reactive theories prefixing has two possible behaviours - waiting for communication and communicating. In the untimed versions of CSP both of those behaviours are relatively simple to define. While waiting to communicate a process ($a \rightarrow Skip$) behaves like $Stop$ and is not refusing to perform the event a ($a \notin \bigcup srefs(slots' \nwarrow slots)$). When the process performs an event - an event is added to the trace history and the process terminates. Two major questions arise while defining prefixing: why do we bother about remembering the history, and why do we need the refusals set?

While in sequential programs remembering history is not necessary, in parallel theories the trace model is used to synchronise interleaved processes. The meaning of the refusals set is slightly more subtle. It allows us to ensure progress of the process. By knowing which events a process is ready to perform we can ensure that it will perform them when the environment is ready.

In slotted-*Circus* in a similar fashion to the definition of deadlock, we have to remember that while waiting time is allowed to pass and empty slots can be generated. Also ensuring progress is not enough because in the timed theory we have to guarantee the maximum urgency rule (events are performed as soon as possible). Both of those aspects are dealt with using the predicate WTC, which describes a behaviour of the prefix process while Waiting To Communicate. The behaviour of $WTC(c)$ first ensures the possibility of generating empty slots using $NOEVTS$ action and then constraining their refusals with the $POSS(c)$ predicate. An important difference between the timed and untimed theories is that there is no longer a single refusals set but a separate one for every time slot and that (while waiting) communication restricts the refusals of all the generated empty slots. This solution allows us not only to observe if a process is able to perform an event, but it also allows to say precisely when. That way we can guarantee that an event will be performed as soon as possible.

The ability to perform a communication after some time, has serious consequences in the definition of prefixing. Because of that, a successful communication is defined as a recording of an event ($EVTSNOW\{c\}$) after a period of waiting (WTC)

$$state' = state \wedge (WTC(c); EVTSNOW\{c\})$$

As explained in [BGW09], while prefixing leaves state unchanged, it keeps the final state unconstrained until termination.

The complete definition of prefixing along with all the necessary healthiness conditions is as follows:

$$[\text{Comm:def}] \quad c \rightarrow Skip \mathrel{\widehat=} CSP1\left(ok' \wedge \mathbf{R3}\left(WTC(c) \lhd wait' \rhd \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array}\right)\right)\right)$$

$$[\text{WTC:def}] \quad WTC(c) \mathrel{\widehat=} POSS(c) \wedge NOEVTS$$

$$[\text{POSS:def}] \quad POSS(c) \mathrel{\widehat=} c \notin \bigcup srefs(slots' \backslash\backslash slots)$$

General prefixing can then be defined in the standard way:

$$[\text{Out:def}] \quad c!e \rightarrow Skip \mathrel{\widehat=} c.e \rightarrow Skip$$

$$[\text{In:def}] \quad c?x \rightarrow Skip \mathrel{\widehat=} \Box_{k:T} \bullet (c.k \rightarrow Skip; x := k)$$

$$[\text{Pfx:def}] \quad comm \rightarrow A \mathrel{\widehat=} (comm \rightarrow Skip); A$$

Here $\Box_{k:T} \bullet P(x)$ is shorthand for $P(k_1) \,\Box\, P(k_2) \,\Box\, \ldots$.

3.9.6 External Choice

External choice ($A \Box B$) allows the environment to choose which of two possible actions A or B runs. For example if we have $(a \rightarrow A) \Box (b \rightarrow B)$, then, if the environment performs a, we see that event occur, followed by an execution of action A. In the event that both sides offer the same event (or the environment is willing to perform events offered by both sides) we get the non-deterministic choice:

$$(a \rightarrow A_1) \Box (a \rightarrow A_2) = a \rightarrow (A_1 \sqcap A_2)$$

The very simple definition of external choice proposed in [HH98], states that either two processes are performing no events and are not terminating ($A \wedge B \wedge Stop$), or one of the process ($A \vee B$) performers an event or terminates ($\neg Stop$). Those two simple cases result in a very simple and easy to use definition.

$$A \Box B =_{CSPdef} A \wedge B \lhd Stop \rhd A \vee B$$

Unfortunately in case of theories with time or state it no longer suffice.

The problem with timed theories is that both A and B can now behave like $Stop$ for some time before

they might offer to perform an event ($A = Wait\ n; P$). Also the environment can decide to resolve the choice for different actions depending on the time. In that situation we either try to predict the future and offer a nondeterministic choice between all possible options, or we chose on the a first-come, first-served basis. Because the first option is unimplementable, a racing condition, considering termination or communication, needs to be added to the definition of external choice. That way the following law holds:

$$((Wait\ t_1; a \to A_1) \Box (Wait\ t_2; a \to A_2)) \setminus \{a\} = Wait\ t_1; (a \to A_1) \setminus \{a\} \qquad for\ t_1 < t_2$$

The external choice operator $A \Box B$ in slotted-*Circus* can behave in three possible ways:
- Time may have passed but no events or termination have occurred: both A and B must agree on refusals, and be behaving like $Stop$. This part of definition (apart from the timed behaviour of $Stop$) is the same as the original definition in CSP.

$$A \wedge B \wedge Stop$$

- An event or termination has occurred, consistent with the corresponding behaviour of A.

$$Choice(A, B)$$

- An event or termination has occurred, consistent with the corresponding behaviour of B.

$$Choice(B, A)$$

In accordance with those three options we get a definition of external choice:

$$[\text{Ext:def}] \qquad A \Box B \mathrel{\widehat=} A \wedge B \wedge Stop \vee Choice(A, B) \vee Choice(B, A)$$

Predicate $Choice(C, R)$ describes the circumstances where an action C is chosen and R refused. The time line of *Choice* can be divided into three periods:
- C and R wait for communication, agree on their refusals and perform no events,
- C performs an event or terminates,
- $Choice(C, R)$ behaves like the chosen action C.

We capture these cases as follows: conjoin R with $NOEVTS$, and follow it sequentially with some "end"-condition that catches behaviour of C in the second and third time section:

$$C \wedge (R \wedge NOEVTS; EndCond)$$

Again there are three possible ending scenarios for the chosen action C:
- performing an (unspecified) event - $IMMEVTS$,
- terminate - $slots \cong slots' \wedge \neg wait'$,
- and diverge - $slots \cong slots' \wedge \neg ok'$.

The last case is very technical as well as counter intuitive, and so it should be explained in more detail. It is possible that one of the processes will start stably, but will diverge while working. In most of the cases this process is harmless but there is a problem when this process behaves like $Stop$:

$$\begin{aligned} DivStop &= \mathbf{R}3 \circ CSP1(NOEVTS \wedge wait') \\ &= Stop \vee \mathbf{R}3 \circ CSP1(NOEVTS \wedge wait' \wedge \neg ok') \end{aligned}$$

While $A \wedge B \wedge Stop$ should be addressing this special case it fails to do that. The reason for that is that $Stop$ stabilises the diverged process.

$$DivStop \wedge Stop = Stop$$

The final definition of choice is as follows:

$$[\mathsf{Choice{:}def}]Choice(C,R) \;\widehat{=}\; CSP2\left(C \wedge \left(\begin{array}{l} R \wedge NOEVTS \\ ; \\ \left(\begin{array}{l} IMMEVTS \vee \\ slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array}\right) \end{array}\right)\right)$$

The definition of **R**3 using $\mathbin{/\!\!/}_R$ rather than $\mathbin{/\!\!/}$ is crucial here, otherwise the following outcome results:

$$(x := 1; a \rightarrow Skip) \Box (x := 2; b \rightarrow Skip) = \text{FALSE} \lhd wait \rhd \ldots$$

As $\mathbin{/\!\!/}$ propagates state changes through, and we are *wait*ing after the assignments, but before a or b is chosen, then the clause $A \wedge B \wedge Stop$ results a contradiction as it asserts, among other things that $x' = 1 \wedge x' = 2$. With $\mathbin{/\!\!/}_R$ used in **R**3, this state information does not appear, so there is no conflict, and once the a or b event is complete, so that we are no longer *wait*ing, then the actual assignment outcome becomes visible.

3.9.7 Parallel Composition

Parallel composition is always the most complicated definition in all CSP based languages. Its complexity arises from the fact that the parallel construct can not simply copy most of the observation variables, but it needs to merge them from the two separate actions. That way the parallel construct has to address every observable aspect: stability, state, time, trace, termination and refusals. What's more, every one of those is as important and it is handled in an unique way.

The basic idea of how parallelism is defined in slotted-*Circus* comes from the UTP book.

$$P \equiv A \mid\mid B \equiv \exists obs_A, obs_B \bullet A[obs_A \backslash obs'] \wedge B[obs_B \backslash obs'] \wedge Merge(obs', obs_A, obs_B)$$

Two parallel processes start with the same set of before-observation (obs) and is working on the local sets of after-observations (obs_A and obs_B). The parallel construct is responsible for merging of the local after-observations into a single set of observations (obs').

The way the $Merge$ function addresses every observable aspect is unfortunately very different from original CSP and needs to be explained separately.

Parallel composition is an operator describing two processes running simultaneously. Because the implementation of parallelism can be very different, in CSP parallelism is parameterised by a set of variables that describes the details of its synchronisation. That way we can describe the whole range of behaviours, starting from fully synchronised parallel $A \mid\mid B$ (where A and B have to agree on every visible event they perform, making them trace equivalent) and finishing with interleaving $A \mid\mid\mid B$ (where A and B have no influence on each others behaviour). This aspect of parallel composition is parameterised by the synchronisation set cs containing all events or event classes on which the two parallel actions have to synchronise. The boundary cases can be defined as:

$$A \mid\mid B \equiv A \underset{E}{\mid\mid} B$$

$$A \mid\mid\mid B \equiv A \underset{\emptyset}{\mid\mid} B$$

Reasoning about parallelism is further complicated by the introduction of a local state. State is a formal representation of memory and so can have many, very different implementation in parallel environments. Because the main focus of CSP based languages is to maintain their algebraic properties, intuitiveness and most of all using communication in order to exchange information between parallel processes, state has been implemented as fully local. Parallel processes run on the separate copies of the state they started in. Once all parallel processes terminate their local finishing states are merged by an unspecified merge function M. For example the subprocess C of $(A \mid\mid B); C$ will be initiated in a state $state_C = M(state'_A, state'_B)$.

While the merge function is not always fully defined in *Circus* it has certain properties that needs to be assumed. First of them is its symmetry.

$$M(state_1, state_2) = M(state_2, state_1)$$

That way we can guarantee the commutativity of parallel composition.

$$(x := 1 \,||\, x := 2) == (x := 2 \,||\, x := 1)$$

Worth noting here is that the final value of x mentioned above can be anything (for example $min = 1$, $max = 2$, $avg = 1.5$ or simply $Unnknown$).

Another property of $M()$, while probably unnecessary in the *CCS* world, ensures the validity of a rarely mentioned law of CSP, that for a deterministic process A, A itself is a unit of parallel composition.

$$A \,||\, A == A$$

That way the following always holds.

$$x := e \,||\, x := e == x := e$$

This property can only be achieved by ensuring that if there are no conflicts on the two parallel options of a state, then the merged result will be equal to them.

$$M(state, state) = state$$

While not being necessary, in slotted-*Circus* state is handled in a much more strict manner and leaves no place for interpretations. That way slotted-*Circus* is being more intuitive and leaves less space for errors. The state merge function is merging all the changes that both process perform on their local states and is undefined (the whole parallel construct diverges) when there are any conflicts. Finally the state is only visible after termination.

$$\begin{array}{l} \textbf{if} \left(\begin{array}{l} s_A \lhd state_A \neq s_A \lhd state \lor \\ s_B \lhd state_B \neq s_B \lhd state \lor \\ s_A \cap s_B \neq \emptyset \end{array} \right) \\ \textbf{then } DIV \\ \textbf{else } wait' \Rightarrow (state' = (s_B \lhd state_A) \oplus (s_A \lhd state_B)) \end{array}$$

In this definition $state$ can be understood as a function betwean variable names and their values. If defined as a set of pairs betwean a variable name and its value ($state = \{(x_1, val_1), (x2, val_2), ...\}$) it results in the following definitions of the given operators:
$\lhd$ operator restricts domain of a relation by the provided set,

$$s \lhd state \mathrel{\widehat=} \{(x, val) \bullet (x, val) \in state \land x \notin s\}$$

$A \oplus B$ joins the two relations,

$$A \oplus B = A \cup B$$

The exact meaning of the provided operators can be redefined, depending on the memory model of the target system.

It is possible to introduce a short hand notation for the parallel construct which does not mention the state synchronisation variables. Instead the assumption is made that the variables will be "well defined".

$$A \underset{cs}{||} B \mathrel{\widehat=}$$

$$A \parallel [s_A \mid \{| cs |\} \mid s_B] B$$

, where s_A and s_B are disjoint sets of all variables changed respectively by A and B.

Stability and termination are both handled in a similar fashion. Process $P = A \underset{cs}{||} B$ is stable when both A and B are stable.

$$ok' = ok_A \wedge ok_B$$

And terminates when both terminate.

$$wait' = wait_A \vee wait_B$$

Finally trace merge is defined using low level definitions on the history model used and needs no special explanation. An interesting aspect of the merge is a control of time synchronisation between the two processes. We have to make sure that we only merge traces recorded at the same time. At the same time we have to allow desynchronising when one of the processes terminates (and no longer records clock ticks). We do that by restricting the time stamp of A an B ($\#slots_A$, $\#slots_B$).

$$(wait_A \Rightarrow \#slots_A \geq \#slots_B) \wedge$$

$$(wait_B \Rightarrow \#slots_B \geq \#slots_A)$$

After considering all the possible use cases we get the following definitions:

[Par:def] $$A \underset{cs}{||} B \mathrel{\widehat=} \exists s_A, s_B \bullet A \parallel [s_A \mid \{| cs |\} \mid s_B] B$$

$$A \parallel [s_A \mid \{| cs |\} \mid s_B] B \mathrel{\widehat=} \exists obs_A, obs_B \bullet$$
$$A[obs_A/obs'] \wedge B[obs_B/obs'] \wedge$$
$$\left(\begin{array}{l} \textbf{if} \left(\begin{array}{l} s_A \lhd state_A \neq s_A \lhd state \vee \\ s_B \lhd state_B \neq s_B \lhd state \vee \\ s_A \cap s_B \neq \emptyset \end{array} \right) \\ \textbf{then} \ \neg ok \wedge slots \preccurlyeq slots' \\ \textbf{else} \left(\begin{array}{l} ok' = (ok_A \wedge ok_B) \wedge \\ wait' = (wait_A \vee wait_B) \wedge \\ (wait' \Rightarrow state' = (state_A - s_B) \oplus (state_B - s_A)) \wedge \\ (wait_A \Rightarrow \#slots_A \geq \#slots_B) \wedge \\ (wait_B \Rightarrow \#slots_B \geq \#slots_A) \wedge \\ VALIDMRG(cs)(slots, slots', slots_A, slots_B) \end{array} \right) \end{array} \right)$$

$$[\text{VMrg:sig}] \quad VALIDMRG : \mathbb{P}Ef((\mathscr{S}\,E)^+)^4 \leftrightarrow \mathbb{B}$$

$$[\text{VMrg:def}] \quad VALIDMRG(cs)(s,s',s_0,s_1) \mathrel{\widehat=} s' \setminus\!\setminus s \in tsync(cs)(s_0 \setminus\!\setminus s, s_1 \setminus\!\setminus s)$$

$$[\text{TSnc:sig}] \quad tsync : \mathbb{P}Ef((\mathscr{S}\,E)^* \times (\mathscr{S}\,E)^*) \leftrightarrow \mathbb{P}((\mathscr{S}\,E)^+)$$

$$[\text{TSnc:sym}] \quad tsync(cs)(s_1,s_2) = tsync(cs)(s_2,s_1)$$

$$[\text{TSnc:nil}] \quad tsync(cs)(\langle\rangle,\langle\rangle) \mathrel{\widehat=} \{\,\}$$

$$[\text{TSnc:one}] \quad tsync(cs)(\langle s\rangle,\langle\rangle) \mathrel{\widehat=} \left\{\ \langle s'\rangle \mid s' \in ssync(cs)(s, snull(sref(s)))\ \right\}$$

$$[\text{TSnc:both}] \quad tsync(cs)\begin{pmatrix} s_1{:}S_1, \\ s_2{:}S_2 \end{pmatrix} \mathrel{\widehat=} \left\{ \begin{array}{l} s'{:}S' \\ \mid s' \in ssync(cs)(s_1,s_2) \wedge \\ \quad S' \in tsync(cs)(S_1,S_2) \end{array} \right\}$$

3.9.8 Hiding

The hiding operator $A \setminus H$ denotes an execution of action A, but with all the events in event-set H hidden. In other words hiding removes all appearances of members of H in the trace. This is achieved using *SHide* function. Another, more subtle, property of hiding is that it marks the scope of communication. In contrast to *CCS* where we talk about point-to-point communication in *CSP* many processes can synchronise with an event. What's more, if two processes are willing to synchronise on a common event they still have to agree with the environment to perform it.

$$a \rightarrow Skip \;||\; a \rightarrow Skip = a \rightarrow Skip$$

For that reason the only way to ensure that an event will be performed is to express that the environment is not interested in the event.

$$a \rightarrow Skip \setminus \{a\} = Skip$$

It is important to note that it is not enough to remove the a event from the trace. We also need to constrain the refusals set to ensure that when a process is ready to perform an event then it will. Without, the following law would hold:

$$a \rightarrow Skip \setminus \{a\} =_{wrong} Stop \sqcap Skip$$

The refusals sets are restricted using.

$$H \subseteq \bigcap srefs(s' \setminus\!\setminus slots)$$

Where H is a set of hidden events.

That way we ensure that the action performs the restricted events and performs them as soon as possible. Without recording of the refusals set for every time cycle it would be impossible to ensure events are being performed in a timely fashion. That would result in the following undesired property:

$$a \rightarrow Skip \setminus \{a\} =_{wrong} Wait\ 0 \sqcap Wait\ 1 \sqcap Wait\ 2 \sqcap Wait\ 3 \sqcap \ldots$$

At the end we add Skip to unconstrain the refusals set of the last slot, so they don't influence the behaviour of following hiding operators.

$$[\text{Hid:def}] \quad A \setminus hidn \mathrel{\widehat=} \mathbf{R}3 \left(\begin{array}{l} \exists s' \bullet A[s'/slots'] \wedge \\ \quad slots' \setminus\!\setminus slots = map(SHide(hidn))(s' \setminus\!\setminus slots) \wedge \\ \quad hidn \subseteq \bigcap srefs(s' \setminus\!\setminus slots) \end{array} \right) ; Skip$$

3.9.9 Events Set Generator

Sometime we want to hide or synchronise on a channel. To do that we introduce a special set notation that will generate all the possible events that implement channel communication.

$$[\text{ChGen:def}] \qquad \{|\, a \,|\} \mathrel{\widehat=} \{\, x \bullet x = a.y \lor x = a \,\}$$
$$\{|\, CH \,|\} \mathrel{\widehat=} \bigcup_{ch \in CH} \{|\, ch \,|\}$$

While in practice use of this notation will require precise typing similar to the one used in FDR, in this work we treat it more like a short hand notation and for convenience we overload it and ignore the typing.

This notation is a part of the *Circus* language [Oli05]p.22.

3.9.10 Timeout

Timeout is modelled as per [She06, p86]

$$[\text{Tout:def}] \qquad A \overset{d}{\rhd} B \mathrel{\widehat=} (A \Box (Wait\ d; int \to B)) \setminus \{\, int \,\}$$

An arguably important problem of timeout is its nondeterministic behaviour when A wants to perform an event after time d. A stricter version of this operator is presented in the next chapter.

3.9.11 Recursion

Recursion in slotted-*Circus* is defined as a least fixed point and in that aspect introduces no changes to the UTP theory of designs. Details about recursion can be found in 2.1.4.

$$[\text{Rec:def}] \qquad \mu X \bullet F(X) \mathrel{\widehat=} \bigsqcap \{\, X \mid X \sqsupseteq F(X) \,\}$$

Chapter 4

Immediate Causality

During this research we found numerous descriptions and names of what surprisingly is the same problem, starting from causal dependence in Timed CSP [1][RR96], immediate causality mentioned in the work on PTCSP [2][Low93], oscillating loops in electronic engineering and ending with prefix closure - a missing healthiness condition in UTP theories[3]. The problem arises when there is no delay between two consecutive events in sequential processes. In slotted-*Circus* this can be described by: $(a \rightarrow b \rightarrow Skip)$, where b can be performed at the measurably same time as a while because they are sequentially composed b has to be proceed by a. For that reason we say that b is caused by a. In electronic engineering this behaviour is known as combinational loops [Chu10], or oscillating loops. It is a situation when a behaviour of hardware logic causes instability on its input, this in return causes instability on its outputs and instability of its behaviour. This can only occur when an input and output of hardware logic are somehow connected and not enough delay is forced on the connection.

> *"Combinational loops are among the most common causes of instability and unreliability in digital designs. In a synchronous design, all feedback loops should include registers. Combinational loops violate synchronous design principles by establishing a direct feedback with no registers. For example, a combinational loop occurs when the left-hand side of an arithmetic expression also appears on the right-hand side. A combinational loop also occurs when you feed back the output of a register to an asynchronous pin of the same register through combinational logic."[Ste05]*

4.1 The Causality Problem in *MSA*

As mentioned before there are currently two available history models for slotted-*Circus* - *MSA* and *CTA*. The main difference between them is that the *MSA* records events that happened within a clock cycle as events that were performed simultaneously while the *CTA* records the order in which they occur. While in the both cases immediate causality is possible, it only introduces complications in the case of *MSA* as described below.

We define two processes that are willing to sequentially perform two events:

$$AB \mathrel{\widehat=} a \rightarrow b \rightarrow Skip$$
$$BA \mathrel{\widehat=} b \rightarrow a \rightarrow Skip$$

[1] "Traces which are equivalent (i.e., are the same except for the permutation of events happening at the same times) are interchangeable. Essentially, this postulates that there can be no causal dependance between simultaneous events: notice that if a process has trace $\langle (t,a),(t,b)\rangle$ then this axiom and axiom 2 show it has trace $\langle (t,b)\rangle$." [RR96]

[2] "There is a non-zero delay between consecutive events in sequential processes, so immediate causality is not allowed. The reader should note that the most recent models of Timed CSP do allow immediate causality. For simplicity we do not allow immediate causality in this thesis." [Low93]

[3] Talk with Jim Woodcock, 2009, York

, and then we try to run them in parallel. Then in the *CTA* world we will get the following law.

$$AB \,||\, BA =_{CTA} Stop$$

Unfortunately in the case of *MSA* the law above does not hold. In order to understand the complications with *MSA* we first have to analyse the possible accepted traces of action AB (we assume that it's stable - $AB \wedge ok$). There are four possible sets of traces (every set is multiplied by the possibility of having empty slots)[4]

- we perform no events

$$slots' \leftthreetimes slots =_{MSA} \langle (\{\,\}, ref_i)^+ \rangle$$

- we perform the a event:

$$slots' \leftthreetimes slots =_{MSA} \langle (\{\,\}, ref_i)^*, (\{a \mapsto 1\}, ref), (\{\,\}, ref_j)^* \rangle$$

- we perform a and b at the same time:

$$slots' \leftthreetimes slots =_{MSA} \langle (\{\,\}, ref_i)^*, (\{a \mapsto 1, b \mapsto 1\}, ref), (\{\,\}, ref_j)^* \rangle$$

- we perform a and b with some time passing between them:

$$slots' \leftthreetimes slots =_{MSA} \langle (\{\,\}, ref_i)^*, (\{a \mapsto 1\}, ref), (\{\,\}, ref_j)^*, (\{b \mapsto 1\}, ref), (\{\,\}, ref_k)^* \rangle$$

If we now perform the same analysis for BA, we will notice that the two processes not only agree on performing no events (as in the case of *CTA*), but can also agree on performing the two events at the same time.

That way the two following sets of traces are accepted by $AB \,||\, BA$:

- we perform no events (behaviour equivalent to $Stop$)

$$slots' \leftthreetimes slots =_{MSA} \langle (\{\,\}, ref_i)^+ \rangle$$

- we perform a and b at the same time:

$$slots' \leftthreetimes slots =_{MSA} \langle (\{\,\}, ref_i)^*, (\{a \mapsto 1, b \mapsto 1\}, ref), (\{\,\}, ref_j)^* \rangle$$

Finally, because the refusals sets can not signal that $AB \,||\, BA$ is willing to perform both a and b while refusing any of those events separately, we loose the maximal-urgency property and so get the following law:

$$(AB \,||\, BA) \setminus \{a,b\} =_{MSA} \sqcap_{n \in N} Wait\ n \sqcap Stop$$

Things get more complicated if we try to "crossover" some information. If we augment the above example to receive and output unchanged information, then we get a similar result, coupled with x being given an

[4] Here a^* denotes an element from a set of all chains of a, $(\varepsilon + a, a^*)$. Plus can be defined as $a^+ = a, a^*$. We overload those notations and use them as generators of possible traces.

$$\begin{array}{ll} & slots = \langle slot_1^*, slot_2, slot_3^+ \rangle \\ \equiv & \\ & slots = \langle slot_2, slot_3 \rangle \\ & \vee slots = \langle slot_1, slot_2, slot_3 \rangle \\ & \vee slots = \langle slot_2, slot_3, slot_3 \rangle \\ & \vee slots = \langle slot_1, slot_2, slot_3, slot_3 \rangle \\ & ... \end{array}$$

arbitrary value.

$$\left(\begin{array}{l}(a?x \rightarrow b!x \rightarrow Skip)\\ \parallel (b?x \rightarrow a!x \rightarrow Skip)\end{array}\right) \setminus \{a,b\} =_{MSA} \sqcap_{n \in N} (Wait\ n; x :=?) \sqcap Stop$$

Here $x :=?$ is shorthand for a non-deterministic choice over assignments to x of any value in its type ($x : T$):

$$x :=? \quad \widehat{=} \quad \sqcap_{k:T} (x := k)$$

If our cross-coupling involves the exchange of conflicting events, any such communication results in a contradiction, so we are left with the deadlock behaviour.

$$\left(\begin{array}{l}(a?x \rightarrow b!x \rightarrow Skip)\\ \parallel (b?x \rightarrow a!(x+1) \rightarrow Skip)\end{array}\right) \setminus \{a,b\} =_{MSA} Stop$$

These examples suggest that slotted-*Circus* "programs" with the *MSA* history model have the ability to decide to deadlock should they be asked to do something contradictory, which raises doubts about its feasibility or computability.

4.2 Timed Prefix

The simplest way to deal with the problems presented above is to remove zero-causality from the language, by providing communication constructs that always take at least one clock cycle to complete. That way any subsequent activity occurs in a clock-cycle after the one in which communication took place. This solution (including notation) is the same as presented in [Low93].

We introduce a new action, *timed prefix*, that forces a number (n) of clock cycles once the communication has occurred:

$$\begin{array}{lcl} a \xrightarrow{n} P & \widehat{=} & a \rightarrow Wait\ n; P \\ a \xrightarrow{0} P & = & a \rightarrow P \end{array}$$

The behaviour of $a \xrightarrow{n} P$ is therefore to wait, allowing the clock to tick, until a occurs, if ever. Once a has happened, then we wait for n clock-ticks before starting P. This captures the behaviour of channel communication in Handel-C, which always requires a clock-tick to complete, and is similar to that of prioritised Timed-CSP. We also see clearly that if $n = 0$ we recover the standard event prefix.

If we build a slotted-*Circus* system using $a \xrightarrow{n} P$ where $n > 0$, then we can avoid the pathological behaviour described in the previous section. In this case, the only way in which more than one communication event can occur in any given clock cycle is as a result of parallel composition involving events that are not required to synchronise. In *MSA*, these are simply gathered into a multi-set history, whilst in *CTA*, we obtain all the possible interleaving traces involving these events.

We can now revisit the earlier pathological examples. Now assuming that $a \rightarrow P$ is replaced by $a \xrightarrow{1} P$ throughout, the order conflict example resolves (in both the *MSA* and *CTA*) to deadlock:

$$(a \xrightarrow{1} b \xrightarrow{1} Skip) \parallel (b \xrightarrow{1} a \xrightarrow{1} Skip) =_{CTA,MSA} Stop$$

Similarly, the crossover examples also result in deadlock:

$$\left(\begin{array}{l} (a?x \xrightarrow{1} b!x \xrightarrow{1} Skip) \\ \| \, (b?x \xrightarrow{1} a!x \xrightarrow{1} Skip) \end{array} \right) \setminus \{a,b\} =_{MSA,CTA} Stop$$

$$\left(\begin{array}{l} (a?x \xrightarrow{1} b!x \xrightarrow{1} Skip) \\ \| \, (b?x \xrightarrow{1} a!(x+1) \xrightarrow{1} Skip) \end{array} \right) \setminus \{a,b\} =_{MSA,CTA} Stop$$

4.2.1 Equivalence of *MSA* and *CTA*

It is interesting to note that in all the above examples where communication takes time there is no different in the outcome with respect to the *CTA* and *MSA* semantic models. The reason for that is the equivalence of the two models when timed communication is enforced. Because events can now be performed at the same time only when they are interleaved, both *CTA* and *MSA* gain special properties. In the case of *MSA*, within the last clock cycle all possible prefixes are accepted. In other words if a process accepts a trace $\langle slot_1, ..., slot_n \rangle$ then it will accept a trace $\langle slot_1, ..., slot_n^* \rangle$, where $slot_n^* \preceq slot_n$ (to simplify the notation we ignore side conditions concerning refusals). For example if a process accepts the trace:

$$\langle slot_1, slot2, (\{a \mapsto 1, b \mapsto 1, c \mapsto 2\}, ref) \rangle$$

Then it will accept (along with all the prefixes of $\langle slot_1, slot2 \rangle$) the following

$$\langle slot_1, slot2, (\{\}, ref_1) \rangle$$
$$\langle slot_1, slot2, (\{a \mapsto 1\}, ref_2) \rangle$$
$$\langle slot_1, slot2, (\{b \mapsto 1\}, ref_3) \rangle$$
$$\langle slot_1, slot2, (\{c \mapsto 1\}, ref_4) \rangle$$
$$\langle slot_1, slot2, (\{a \mapsto 1, b \mapsto 1\}, ref_5) \rangle$$
$$\langle slot_1, slot2, (\{a \mapsto 1, c \mapsto 1\}, ref_6) \rangle$$
$$\langle slot_1, slot2, (\{b \mapsto 1, c \mapsto 1\}, ref_7) \rangle$$
$$\langle slot_1, slot2, (\{c \mapsto 2\}, ref_8) \rangle$$
$$\langle slot_1, slot2, (\{a \mapsto 1, b \mapsto 1, c \mapsto 1\}, ref_9) \rangle$$
$$\langle slot_1, slot2, (\{a \mapsto 1, c \mapsto 2\}, ref_{10}) \rangle$$
$$\langle slot_1, slot2, (\{b \mapsto 1, c \mapsto 2\}, ref_{11}) \rangle$$

This property is called a strong prefix healthiness and will be explained in detail in the next section.

In the case of *CTA* we also get a very strong property where if a trace $\langle slot_1, ..., slot_n \rangle$ is accepted then $\langle slot_1^*, ..., slot_n^* \rangle$ is accepted as well, where $slot_i^*$ has a history part equal to any permutation of the history of $slot_i$. For example if a process accepts the trace:

$$\langle (< a, b >, ref_1), (\langle c, d, d \rangle, ref_2) \rangle$$

Then it will accept the following:

$$\langle(<a,b>,ref_1),(\langle c,d,d\rangle,ref_2)\rangle$$
$$\langle(<a,b>,ref_1),(\langle d,c,d\rangle,ref_2)\rangle$$
$$\langle(<a,b>,ref_1),(\langle d,d,c\rangle,ref_2)\rangle$$
$$\langle(<b,a>,ref_1),(\langle c,d,d\rangle,ref_2)\rangle$$
$$\langle(<b,a>,ref_1),(\langle d,c,d\rangle,ref_2)\rangle$$
$$\langle(<b,a>,ref_1),(\langle d,d,c\rangle,ref_2)\rangle$$

Once we have those two properties we can link the *MSA* and *CTA* histories. The set of *CTA* histories generated from *MSA* is simply the set of all possible permutations of the corresponding *MSA* history. Because all the possible permutations of accepted *CTA* history are always accepted, the link going in the other direction is always possible. Also strong prefixing of the corresponding *MSA* history, guarantees that any prefix of any permutation of the *CTA* trace will have its corresponding trace in the *MSA*.

4.3 Prefix Closure

While UTP along with the denotational semantics of CSP, is well known for over ten years, there are are still some aspects of the theory that have not yet been addressed. One of them, is a missing healthiness condition [5] known as the Prefix Healthiness [Ros98]. The Prefix Healthiness is a property of all the CSP based languages. It states that if a process accepts a trace then it should also accept all its prefixes. Because in all the CSP based theories we can communicate only one event at a time and every definition of prefixing always has an option of accepting an empty trace, this condition is never violated. Also, because there seems to be no need for this property in proofs in theories other than in slotted-*Circus*, no attention has been paid formulating the Prefix Healthiness condition in UTP .

There are two situations in slotted-*Circus* when the lack of the Prefix Healthiness proves problematic. The first is a need for a property stating that any process accepts an empty trace. We need this in the case of reasoning about external choice.

$$Choice(C,R) \;\widehat{=}\; CSP2\left(C \wedge \left(\begin{array}{l} R \wedge NOEVTS \\ ; \\ \left(\begin{array}{l} IMMEVTS \vee \\ slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array}\right) \end{array}\right)\right)$$

If we look closely at the *Choice* definition we will notice a hidden assumption, necessary for it to work. We assume that there exists an empty trace, accepted by the refused action R so that there are always situations when $R \wedge NOEVTS$ holds true. This property can be described as:

Assuming no time travel is allowed ($R = \mathbf{R}1(R)$) and R is not a Miracle ($R = R \wedge ok$), then

$$[\text{NullPrefix:Assumption}] : p.143 \qquad \exists obs' \bullet Q \wedge ok \wedge \neg wait \wedge slots \cong slots'$$

The second situation when we need prefix healthiness is when we want to reason about zero-causality, which we will explain later.

[5] Talk with Jim Woodcock, York 2009

4.3.1 Violating Prefix Closure

An interesting aspect of the *MSA* history model is that it is the only known existing theory in CSP family that is able to violate prefix closure. As shown before in the case of:

$$a \rightarrow b \rightarrow Skip \,||\, b \rightarrow a \rightarrow Skip\ ,$$

while performing a or b alone is impossible, *MSA* allows us to perform both a and b at the same time. As we can see this undesired situation is directly connected with zero-causality.

4.3.2 Weak and Strong Prefix Healthiness

Even more interesting is the fact that the exact properties of prefix closure are not as obvious in the case of *MSA* as they are for *CTA*, *Circus*, or CSP. If we consider the following *MSA* history:

$$hist_{MSA} = \{a \mapsto 1, b \mapsto 1\}$$

Then we can imagine a case when a is sequentially composed with b and so the list of accepted prefixes will be

$$\{\}$$
$$\{a \mapsto 1\}$$

Unfortunately this is not a complete list of prefixes of $hist_{MSA}$. In fact we can list three possible prefixes of it:

$$\{\}$$
$$\{a \mapsto 1\}$$
$$\{b \mapsto 1\}$$

While prefixing is unique for sequences it is not unique in the bag world. For that reason we define two versions of prefix healthiness for *MSA*. The strong one where we demand acceptance of all the possible prefixes, and the weak one, where for any accepted trace we accept at least one immediate prefix of the trace.

An example of a slotted-*Circus* process that violates the strong prefix healthiness while being weak prefix healthy is $a \rightarrow b \rightarrow Skip$ which while accepting $\langle(\{a \mapsto 1, b \mapsto 1\}, ref_1)\rangle$ can never accept $\langle(\{b \mapsto 1\}, ref_2)\rangle$

4.3.3 Defining Prefix Closure

The results presented in this section is part of ongoing research on Prefix Healthiness for slotted-*Circus* and UTP theories in general, performed together with Andrew Butterfield and Jim Woodcock. The definitions presented in this section have not yet been verified and should be treated with caution.

Currently prefix closure for slotted-*Circus* for the *CTA* history model is defined as:

[isPH:CTA:def]

$$isPH(P) \mathrel{\widehat{=}}_{CTA} \forall slots', slots, slots_p \bullet \left(\begin{array}{c} (P \wedge (slots \preccurlyeq slots_p \preccurlyeq slots') \wedge ok') \\ \Downarrow \\ (\exists obs' \bullet P \wedge slots_p \cong slots' \wedge ok') \end{array} \right)$$

$isPH(P)$ is true when for all prefixes of the accepted final trace ($slots'$), there exists a situation ($\exists obs'$), when the prefix ($slots_p$) is accepted. It is important to note here the use of $\cong$. When we say "all prefixes" we do not mean all the possible values of refusals set, and so we want the refusals to be existentially quantified.

The same definition can be used to define the strong prefix healthiness for the *MSA* history model.

$$[\mathsf{isSPH{:}def}]$$

$$isSPH(P) \mathrel{\widehat{=}} \forall slots', slots, slots_p \bullet \left(\begin{array}{c} (P \wedge (slots \preccurlyeq slots_p \preccurlyeq slots') \wedge ok') \\ \Downarrow \\ (\exists obs' \bullet P \wedge slots_p \cong slots' \wedge ok') \end{array} \right)$$

Unfortunately defining the weak prefix healthiness is not as easy. To do that we have to start by defining a notion of immediate prefix.

Because $front(slots)$ is a partial function and because the last slot can be empty we break the definition into three cases:

$$[\mathsf{ImPref{:}def}]$$

$$ImPref(slots) \mathrel{\widehat{=}}_{SC}$$

$$\left\{ \begin{array}{ll} \{\,\} & , where\ slots = \langle snull \rangle \\ \{\, front(slots) \,\} & , else\ if\ last(slots) = snull \\ \left\{ \begin{array}{l} front(slots)^\frown \langle (hist_1, ref_1) \rangle \bullet last(slots) = (hist_2, ref_2) \\ \quad \wedge hist_1 \preceq hist_2 \wedge \sharp hist_1 = (\sharp hist_2 - 1) \end{array} \right\} & , else \end{array} \right.$$

.

$ImPref$ is a function on $slots$ that returns a set of all the immediate prefixes of $slots$. Immediate prefix of a trace is a trace that is one event or clock tick "shorter". If the trace is empty then it has no prefixes. If the last slot of $slots$ is an empty slot ($last(slots) = snull$) then the only immediate prefix of this trace is a prefix without the empty slot ($front(slots)$). Else if the last slot of $slots$ is non empty the we are interested about all the prefixes of the last slot that are only one event smaller ($\sharp hist_1 = (\sharp hist_2 - 1)$). $\sharp$ function is not currently defined for slotted-*Circus* history models. But will be added once the proposed definition has been verified.

Once we are able to generate the set of all the immediate prefixes of a trace, we can define the weak prefix healthiness.

$$[\mathsf{isWPH{:}def}]$$

$$isWPH(P) \mathrel{\widehat{=}} \forall slots', slots, slots_p \bullet$$

$$\left(\begin{array}{c} (P \wedge (slots \preccurlyeq slots_p \preccurlyeq slots') \wedge ok' \wedge \neg (slots \cong slots_p)) \\ \Downarrow \\ ((\exists obs' \bullet P \wedge slots_p \cong slots' \wedge ok') \Rightarrow (\exists obs' \bullet P \wedge slots' \in ImPref(slots_p) \wedge ok')) \end{array} \right)$$

A process is weak prefix healthy if for any accepted trace, if it accepts its prefix, then it accepts at least one of the immediate prefixes of the accepted prefix. In other words, if we analyse the recursion hidden in the definition, a weak prefix healthy process accepts at least one of the immediate prefixes of any accepted traces.

Using the defined prefix tests we can try to formulate prefix predicate transformers

$$\begin{array}{rrl} [\mathsf{PH{:}def}] & PH(P) \mathrel{\widehat{=}} & CSP1(isPH(P) \wedge P) \\ [\mathsf{SPH{:}def}] & SPH(P) \mathrel{\widehat{=}} & CSP1(isPH(P) \wedge P) \\ [\mathsf{WPH{:}def}] & WPH(P) \mathrel{\widehat{=}} & CSP1(isWPH(P) \wedge P) \end{array}$$

According to the above definitions $PH(P)$ diverges when not prefix healthy.

4.4 Ensuring the Safety of *MSA*

If we list all the slotted-*Circus* examples that were used to talk about violation of the weak or strong prefix healthiness, then we will notice that they overlap with the examples presented in the discussion about the immediate causality. In fact the two topics are directly related and we can conclude them with two statements.

The first one is based on an earlier observation that cross communication violates the prefix closure.

Observation 1: *A process performing cross communication, violates prefix closure (weak prefix healthiness).*

Another observation explains how we can violate prefix healthiness with prefix healthy actions. Because strong and weak healthiness differs only in case of *MSA* the following observation can only be adapted to slotted-*Circus* with the *MSA* history model. This also explains why only in *MSA* we are forced to discuss the prefix closure and the cross communication.

Observation 2: *Only a processes that is weak healthy but not strong healthy (unsafe) can be forced by the environment to violate the prefix closure.*

The simplest conclusion from the two observations is that we can ensure prefix closure (weak prefix healthiness) by enforcing strong prefix healthiness. And the easiest way to do that is by enforcing timed communication.

Observation 3: *Every action restricted to timed communication is strong prefix healthy.*

Unfortunately untimed communication and immediate causality are powerful modelling tools. They are especially important when we want to describe time efficient hardware 7.1.2, 7.3 or when we want to reason about non-standard communication 7.2. For that reason it is important to learn how can we use a weak-healthy processes safely.

We can do that by a careful analysis of unsafe (weak but not strong prefix healthy) actions, and reducing the scope of unsafe actions using hiding. When analysing an action we bear in mind the observation that unsafe processes can be forced by the environment to violate prefix closure. For that reason we have to check prefix closure whenever an unsafe action is used with the parallel operator. An action that violates prefix closure can not be "repaired" and can never be allowed in the specification. This is especially important when we try to control the scope of unsafe behaviour.

Observation 4: *Hiding can "healthify" a prefix unhealthy process.*

We can make an unsafe action safe by hiding some of its events. By restricting the history we can make any process strong prefix healthy. In case of the weak healthy actions this is a desired property.

$$\begin{aligned} QuickFunction &\mathrel{\widehat{=}} internal?x \rightarrow x := f(x); internal!x \rightarrow Skip \\ Process(x) &\mathrel{\widehat{=}} internal!x \rightarrow internal?x \rightarrow Work(x) \end{aligned}$$

In the above example both *Process* and *QuickFuction* are unsafe. But (assuming $Work(x)$ is safe - it is only using timed communication) we can make them safe if we restrict the range of their unsafe behaviour.

$$SafeProcess(x) \mathrel{\widehat{=}} (Process(x) \underset{\{|internal|\}}{||} QuickFunction) \setminus \{| internal |\}$$

By defining the *SafeProcess* in a careful way we can show that it is equivalent to:

$$SafeProcess(x) =_{MSA} x := f(x); Work(x)$$

While the risk of using an unsafe definition seems unnecessary, we can imagine a situation where $f(x)$ is implemented using a number of parallel logic circuits and for the purpose of efficiency we do want $f(x)$ to be computed within a clock cycle.

It is Important to note that whenever hiding is used we have to make sure that it is not used on a process that violated the prefix closure.

Observation 5: *Hiding should never be applied to a process that is not weak healthy.*

Using hiding on a process that is not weak prefix healthy can result in a perfectly healthy but unpredictable behaviour.

$$\left(\begin{array}{l} (a?x \rightarrow b!x \rightarrow Skip) \\ \parallel (b?x \rightarrow a!x \rightarrow Skip) \end{array} \right) \setminus \{a,b\} =_{MSA} \sqcap_{n \in N} (Wait\ n; x :=?) \sqcap Stop$$

4.4.1 Using Untimed Communication

The discussion presented above along with the tools to control the safety of an action is not precise enough to be built into a tool support. For that reason in the authors opinion untimed communication should only be used when necessary and should be used within a limited scope of predefined macros. The rest of the specification should always use the timed communication.

The macros should be proven to be equivalent to an action that uses only timed communication. If that is impossible a careful explanation along with precise restrictions on the environment must be provided and the macro should be simple enough to be obviously safe.

4.5 Conclusions

In this chapter what seems to be an undesired phenomenon of immediate causality of two events (section 4.1) is discused . While immediate causality reveals itself only in case of the *MSA* and never in *CTA* history model, it can be argued (section 5.5) that for the same reasons only *MSA* could be used as a backbone history model for theories with a notion of priority. Two ways of handling immediate causality are presented. The first one (section 4.2) is a very simple solution that restricts the prefix operator, so that communication/event execution always takes some (long enough to be recorded) time. The second solution (subsection 4.4.1) is a hybrid approach, that suggests the use of the untimed prefix only when necessary and hiding it from the user of the language behind predefined macros. In section 4.3 a missing healthiness condition in slotted-*Circus* - the prefix closure is presented. It is also shown that it is crucial in restraining the causality problem and ensuring safety of processes defined with the untimed communication.

The results presented in this chapter are relatively novel and should be cross checked in other similar theories. Also because untimed communication proved to be very useful in the case studies (chapter 7), based on definitions of weak and strong prefix healthiness ([isSPH:def]:p. 47, [isWPH:def]:p. 47) more practical ways of ensuring safety should be found.

Chapter 5

Priority

5.1 Introduction

In this chapter we focus on the notion of priority described in 2.2. We start by providing an intuitive representation of priority along with examples illustrating the most important conclusions of this work (Sec. 5.2). Then we discuss the algebraic properties of prioritised choice and based on them we provide the definition of the prioritised choice operator ($\overleftarrow{\Box}$) (Sec. 5.3). The definition of the new operator is followed by a discussion about the new healthiness condition (Sec. 5.3.1), and the necessary changes to the slotted-*Circus* theory (Sec. 5.3.2). We introduce an improved language constructs that can be defined using the prioritised choice operator (Sec. 5.3.2). Finally, we provide a formal arguments for infeasibility of priority in CSP (and any other untimed theory based on it) (Sec.5.4) and slotted-*Circus* with the *CTA* history model (Sec. 5.5).

5.2 Intuitive Example

Maria wants to get married and she currently has only two possible candidates - Bob and Charles. Because Bob studies formal methods Maria prefers him to Charles who is a fitness instructor. Unfortunately Maria can not be sure if her feelings are mutual and doesn't take "No" for an answer (with the hope that they can change their minds later). Another restriction is that Maria can have only one husband. Maria's task is if possible to get married, and if she has a choice then to get married to Bob.

5.2.1 Case 1: The need for a deadline

After asking Bob and Charles to get married, Charles agreed while Bob said that he will have to think about it. Maria has two options now: wait for Bob to make up his mind, or to get married with Charles. Unfortunately getting married with Charles straight away would not leave any chance for the preferred Bob to make up his mind. In fact marrying Charles would mean that Maria married the first candidate to give her an answer and that her preference was irrelevant. For that reason Maria decides to wait. Because there is always a chance that Bob will finally say "Yes" (we assume that Maria can not predict the future), Maria needs to set a deadline after which she will no longer accept Bob. Without the deadline asking Charles would be unnecessary because Maria would wait for ever for Bob to change his mind in any case.

This example represents the reasons why priority can only be defined in theories that have a notion of time. In fact the most important thing here is the ability to provide a precise deadline for the choice to be made. Formally this conclusion is used to show infeasibility of prioritised choice in *CTA* in Section 5.4.

5.2.2 Case 2: Problems with rolling back

Maria could always commit her feeling to one of her candidates while still waiting for her preferred options to change her mind. Unfortunately, while it is possible to change your mind after a few dates, with passing events rolling back is increasingly difficult (shared mortgage, engagement, wedding), and at one point becomes impossible (kids). Because of that we can not expect that Maria will always marry Bob if it's ever possible (unless she can predict the future and so predict that Bob will one day change his mind).

At the other hand the absolute inability to change her mind means that Maria commits to get married whenever she think that someone is a worthy candidate. This implies that Maria should only try to convince her candidates sequentially. Because of her preference that would mean to start meeting Bob first and when he gives no positive answer for a while then to try with Charles.

This solution, while feasible does not scale up to to larger number of candidates. We can think of Maria looking for a husband on the internet via a dating website, maintaining contact with only one candidate at a time and always trying to get married to the best candidates. This would result in a very long decision making process and could prove completely ineffective if Maria was not a top candidate herself.

5.2.3 Case 3: Causality loops

Finally we can imagine a special case in which Maria will not be able to make a decision or her decision will end up being random. Let's assume that Bob, Maria and Charles live in a desert island and the decision that Bob makes is dependent on the actions of Charles. One day Bob decided that he will only think of getting married if Charles will (assuming that they were on a desert island, we conclude that at the end Bob was not very good in formal methods). Maria after asking both of her candidates got a "Yes" for Charles and "No" from Bob. Unfortunately, once Bob found out that Charles is getting married he changed his mind. That caused Maria to change her mind and choose her preferred Bob. Because Charles was no longer getting married this in consequence made Bob change his mind. This causality loop will make Maria change her choices until the deadline for the decision is reached. If she will be lucky her final decision will be to marry Charles, but she might be unfortunate and make her mind on Bob (which because of hidden restrictions is impossible to be married with). Of course Maria might be able to analyse the behaviour of Bob but causality links can sometimes be difficult/impossible to draw. The simple solution for this case is to ensure that the causality loop takes more time to be executed than the decision itself. Maria could enforce that by making her plans with Charles secret until the final decision is made (that way Bob will not find out in time that Charles is getting married).

This example represents the immediate causality issue in slotted-*Circus* and it best fits the example presented in the discussion of priority in slotted-*Circus* with the *CTA* history model and no timing restrictions 5.5.

5.2.4 How to find a husband

Based on the above examples we can try to draw a decision making process for Maria. She should approach all of her candidates simultaneously and start her negotiations/try to convince them as soon as possible. That way she will ensure effectiveness and will maximise the results (choosing most preferable option) within a reasonable time. She should have a deadline for the final decision. Until the deadline she shouldn't do anything that will be impossible to undo (unless she can marry her top candidate, then waiting is unnecessary), this will allow her to change her mind when a better candidate will become available. Finally she should take steps to avoid the change of mind of the candidate that she commits to (the worst case is when she commits and he changes his mind just after without committing himself). While this can be addressed in many ways, our solution suggests that when the final commitment is being negotiated Bob and Maria should have limited (delayed) ability to communicate with other people that could influence their choice. An example of where they shouldn't do it could be a large family dinner, where both dowry and bride price is part of the tradition.

5.3 Defining Priority

As mentioned before priority is an implementation of external choice. For that reason their behaviour is in most cases identical. When only one event is available then it will be performed. When events are available in different clock cycles then the choice is being resolved on a first-come first-served basis.

$$(Wait\ n; a \rightarrow A \,\Box\, (Wait\ m; b \rightarrow B)) \setminus \{a,b\} = Wait\ n; A \setminus \{a,b\}$$

Here $n < m$.

Because of the use of hiding the a event will be performed immediately after n clock ticks. Because $n < m$ at the time the b event will not be available so the left hand side of the external choice will be chosen. Similar reasoning applies to the prioritised choice.

$$(Wait\ n; a \rightarrow A \stackrel{\leftarrow}{\Box} (Wait\ m; b \rightarrow B)) \setminus \{a,b\} = Wait\ n; A \setminus \{a,b\}$$

$$(Wait\ n; a \rightarrow A \stackrel{\rightarrow}{\Box} (Wait\ m; b \rightarrow B)) \setminus \{a,b\} = Wait\ n; A \setminus \{a,b\}$$

Here $n < m$.

The difference between the two operators can only be observed when both of the available options offer an event (or termination) at the same time. In that case external choice is nondeterministic, while the prioritised choice refines this behaviour by making the low priority option unavailable.

$$(Wait\ n; a \rightarrow A \,\Box\, (Wait\ n; b \rightarrow B)) \setminus \{a,b\} = Wait\ n; (A \sqcap B) \setminus \{a,b\}$$

$$(Wait\ n; a \rightarrow A \stackrel{\leftarrow}{\Box} (Wait\ n; b \rightarrow B)) \setminus \{a,b\} = Wait\ n; A \setminus \{a,b\}$$

Because the prioritised choice is a refinement of the external choice, in order to define it we only have to strengthen the definition of external choice by restricting some of its behaviours. As we can see in the example above the part of the definition of external choice that will need restricting is deciding when to choose the low priority option (LP).

$$HP \stackrel{\leftarrow}{\Box} LP \mathrel{\widehat{=}} (HP \wedge LP \wedge Stop) \vee Choice(HP, LP) \vee WeakChoice(LP, HP)$$

Where,

$$WeakChoice(LP, HP) \mathrel{\widehat{=}} Choice(LP, HP) \wedge RestrictingCondition$$

Before we define the *WeakChoice* we first want to precisely describe all the differences between prioritised and external choice, which we do by stating some laws that we expect to be obeyed by prioritised choice.

$$(1) \qquad (a \rightarrow HP \stackrel{\leftarrow}{\Box} b \rightarrow LP) \setminus \{a,b\} = HP \setminus \{a,b\}$$

When both high and low priority actions are ready to perform an event at the same time then the high priority option is chosen. Similar law along with a proof can be found in the Appendix [hiding:PriChoice]:p. 141.

$$(2) \qquad a \rightarrow HP \stackrel{\leftarrow}{\Box} a \rightarrow LP = a \rightarrow HP$$

When the high and low priority option are willing to perform the same event, then we can resolve the choice for the benefit of the higher priority option. This occurs because once the environment will be ready to perform

the a event, it will be at the same time for both of the options, and so the first law will apply.

$$(3) \qquad Skip \overset{\leftarrow}{\Box} LP = Skip$$

$Skip$ describes a process that terminates immediately and termination is treated as an event). The low priority option can not offer any event sooner than that and so, can be removed.

We also want to bear in mind the fourth law below in order not to over-constrain the low priority option. It states that we can not choose the high priority option if we are not sure about the exact timing of when the events a and b will be available. Law (4) also highlights the difference between laws (1) and (2), the importance of hiding and why (2) is not just a special case of (1).

$$(4) \qquad a \rightarrow HP \overset{\leftarrow}{\Box} b \rightarrow LP \neq a \rightarrow HP$$

Finally law (5) (which is also a law of external choice) reminds us that laws (1)-(4) are not only applicable when events/ termination are available in the first clock cycle, but also apply after the clock has ticked without any events taking place.

$$(5) \qquad (Wait\ n; HP) \overset{\leftarrow}{\Box} (Wait\ n; LP) = Wait\ n; (HP \overset{\leftarrow}{\Box} LP)$$

We start from the definition of *Choice* and the law (3).

$$Choice(P,S) \mathrel{\widehat{=}} CSP2(P \wedge \left(\begin{array}{l} (S \wedge NOEVTS); \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))$$

$$(3) \qquad Skip \overset{\leftarrow}{\Box} A = Skip$$

In CSP theories termination is always treated like an event, but in our semantics it is not recorded in the history and so it needs special care while writing any definition. Law (3) states that if the High Priority (HP) option terminates, then the Low Priority option (LP) can not be chosen. Therefore we want to restrict the *WeakChoice* so that the LP can only by chosen if the HP did not terminate ($HP \wedge wait'$).

$$CSP2(LP \wedge \left(\begin{array}{l} (HP \wedge NOEVTS \wedge \mathbf{wait'}); \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))$$

The second law states that when both HP and LP are ready to perform the same event then LP can be ignored. While we do not know when the referred event will be performed. We do know that the environment will be willing to perform it at the same time for both sides of the choice operator. For that reason we can be sure that the HP action will proceed with it.

$$(2) \qquad a \rightarrow HP \overset{\leftarrow}{\Box} a \rightarrow LP = a \rightarrow HP$$

The reason why this law is not addressed by (1) is because (1) depends on hiding to enforce the the racing conditions. That way only ($a \rightarrow HP \overset{\leftarrow}{\Box} a \rightarrow LP \setminus \{a\} = HP \setminus \{a\}$) is a special case of (1).

(2) is addressed by changing: $IMMEVTS$,into:

$$\exists E \bullet FSTEVTS(E) \wedge E \neq \emptyset \wedge E \subseteq sref(last(slots))$$

That way LP can only perform an event which is not accepted at the moment by HP.

$$CSP2(LP \wedge \left(\begin{array}{l} (HP \wedge NOEVTS(slots, slots') \wedge wait') \\ ; \\ \left(\begin{array}{l} (\exists \mathbf{E} \bullet \mathbf{EVTSNOW}(\mathbf{E})(\mathbf{slots}, \mathbf{slots}') \wedge \mathbf{E} \neq \emptyset \wedge \mathbf{E} \subseteq \mathbf{sref}(\mathbf{last}(\mathbf{slots}))) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))$$

Finally we can address the most important law of priority - (1). According to this law, LP can only be refused when HP is willing to perform an event and environment is demanding the event to be performed in the first possible time slot. This behaviour is very similar to the waiting option of the prefix operator:

$$CSP1(ok' \wedge \mathbf{R}3(WTC(c) \wedge wait'))\setminus\{c\} = Miracle^{1}$$

that gets refused (is falsified within the slotted-*Circus* lattice) when the environment decides that an event needs to be performed in the first time slot. For that reason we want to copy the mechanism of interaction between hiding and prefix, by not refusing events that are not refused by HP at the time when the choice is being resolved. We do that by:

$$CSP2(LP \wedge \left(\begin{array}{l} (HP \wedge NOEVTS \wedge wait'); \\ \left(\begin{array}{l} \mathbf{sref}(\mathbf{head}(\mathbf{slots}' \backslash\backslash \mathbf{slots})) \subseteq \mathbf{sref}(\mathbf{last}(\mathbf{slots})) \wedge \\ \left(\begin{array}{l} (\exists E \bullet FSTEVTS(E) \wedge E \neq \emptyset \wedge E \subseteq sref(last(slots))) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right))$$

That way we gain the following law.

$$WeakChoice(LP, a \rightarrow HP)\setminus\{a\} = Miracle$$

Proof [WeakChoice:reduce]:p. 148

[1] In the slotted-*Circus* lattice *Miracle* represents logical *false* and within the semantics has very similar properties. For example in slotted-*Circus* *Miracle* is a unit of disjunction ($Miracle \vee CSP1(P) = CSP1(P)$). That way proving one of the possible behaviours being equal to *Miracle* corresponds to falsifying part of a logical equation.

With the ability to remove *WeakChoice* we can now prove (1).

$$
\begin{array}{ll}
 & (a \to P \overleftarrow{\Box} Q) \setminus \{a\} \\
\equiv & \text{“ [pExt:def]:p. 56 ”} \\
 & \left(\begin{array}{l} a \to P \wedge Q \wedge Stop \\ \vee Choice(a \to P, Q) \\ \vee WeakChoice(Q, a \to P) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ hiding distributes over disjunction ”} \\
 & (a \to P \wedge Q \wedge Stop) \setminus \{a\} \\
 & \vee Choice(a \to P, Q) \setminus \{a\} \\
 & \vee WeakChoice(Q, a \to P) \setminus \{a\} \\
\equiv & \text{“ reducing the } \textit{WeakChoice} \text{ option ”} \\
 & (a \to P \wedge Q \wedge Stop) \setminus \{a\} \\
 & \vee Choice(a \to P, Q) \setminus \{a\} \\
 & \vee Miracle \\
\equiv & \text{“ performing an event and not performing it } (Stop) \text{ is impossible [CommAndStop:reduce]:p. 142 ”} \\
 & Miracle \\
 & \vee Choice(a \to P, Q) \setminus \{a\} \\
 & \vee Miracle \\
\equiv & \text{“ Property of CSP1 ”} \\
 & Choice(a \to P, Q) \setminus \{a\} \\
\equiv & \text{“ Property of } \textit{choice} \text{ [Choice:perform]:p. 144 ”} \\
 & P \setminus \{a\}
\end{array}
$$

The complete proof of this law can be found in the appendix [hiding:PriChoice]:p. 141.

5.3.1 Prioritised Choice Definition

After introducing the new healthiness condition we can finally present a complete definition of prioritised choice:

$$
\begin{array}{rl}
\text{[pExtAsym:def]} & L \overrightarrow{\Box} H \mathrel{\widehat{=}} H \overleftarrow{\Box} L \\
\text{[pExt:def]} & H \overleftarrow{\Box} L \mathrel{\widehat{=}} H \wedge L \wedge Stop \vee Choice(H, L) \vee WeakChoice(L, H) \\
\text{[pExt:def:WeakChoice]} & WeakChoice(L, H) \mathrel{\widehat{=}} CSP2(L \wedge \\
 & \left(\begin{array}{l} \left(H \wedge NOEVTS \wedge wait' \right); \\ \textbf{PRI} \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E) \\ \wedge E \neq \emptyset \wedge E \subseteq sref(last(slots)) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))
\end{array}
$$

In the appendix we provide proofs of the following laws of priority:

$$
\begin{array}{rl}
\text{[hiding:PriChoice]} : p.141 & (a \to P \overleftarrow{\Box} Q) \setminus \{a\} \equiv P \setminus \{a\} \\
\text{[pECimplementsEC]} : p.157 & (P \overleftarrow{\Box} Q) \sqsubseteq (P \Box Q)
\end{array}
$$

5.3.2 Changes in slotted-*Circus*

From the beginning priority in slotted-*Circus* was meant to be a small extension to the existing language rather than a completely new one. While defining a new operator was not enough, we still managed to reason about priority without changing the trace model of slotted-*Circus* , redefining other operators or introducing new observation variables. What had to be changed in slotted-*Circus* was a new healthiness condition and making hiding CSP3 healthy.

PRI healthiness condition

The reason why we need a new healthiness condition is that the prioritised choice operator uses refusals sets in a new way and expands their functionality. Therefore we have to make sure that the information stored in it is properly propagated and that whenever a process terminates *CSP*4 no longer completely unconstrains its refusals sets. In more detail we have to make sure that whenever a process is not refusing an event then this information will not be omitted by the following processes.

$$\begin{array}{l} ok' \wedge a \notin sref(last(slots')); \mathbf{PRI}(P) \\ \Downarrow \\ P = P \wedge a \notin sref(head(slots' \backslash\backslash slots)) \end{array}$$

We enforce that by introduction of a new healthiness condition.

$$[\mathsf{PRI{:}def}] \qquad \mathbf{PRI}(P) \mathrel{\widehat=} P \wedge (ok \Rightarrow sref(head(slots' \backslash\backslash slots)) \subseteq sref(last(slots)))$$

The proper propagation of information stored in the refusals set is necessary for the correct behaviour of priority. As mentioned before, prioritised choice is "marking" the refusals set of the low priority option with the refusals of high priority options.

$$\begin{array}{rl} & ((a \to Skip) \overset{\leftarrow}{\Box} (b \to Skip)) = a \to Skip \,\Box\, MarkRefs\{a\}(b \to Skip) \\ \text{where,} & MarkRefs\{a\}(P) \mathrel{\widehat=} \mathbf{R}3(CSP1(a \notin sref(head(slots' \backslash\backslash slots)))) \wedge P \end{array}$$

This property works correctly even without any changes to slotted-*Circus*. We can imagine a situation when the prioritised choice is sequentially composed with another CSP3 healthy process.

$$((a \to Skip) \overset{\leftarrow}{\Box} (b \to Skip)); CSP3(C)$$

If we now expand the behaviour of the low priority option we get:

$$MarkRefs\{a\}(b \to Skip); CSP3(C) =_{SC} MarkRefs\{a\}(b \to Skip); Skip; C =_{SC} b \to C$$

, because *Skip* unconstrains refusals set of the last slot of the previous process and so removes the refusals restrictions/"markings" of the low priority option.

PRI healthiness, by ensuring propagation of just enough information about refusals set, is protecting us from this situation.

$$\begin{array}{rll} MarkRefs\{a\}(b \to Skip); CSP3(C) & =_{pSC} & MarkRefs\{a\}(b \to Skip); \mathbf{PRI}(Skip); \mathbf{PRI}(C) \\ & =_{pSC} & MarkRefs\{a\}(b \to C) \end{array}$$

Impact of PRI on slotted-*Circus*

In slotted-*Circus* similar to any other CSP based language only one operator can change the set of acceptable traces depending on refusals sets - hiding. Because **PRI** healthiness only introduces changes to refusals sets and hiding has been adapted to work correctly with those changes the **PRI** healthiness condition is neutral with respect to slotted-*Circus*. This means that all the proofs that do not include the new definition of hiding or $Skip$ can be left unchanged. Also all the laws of slotted-*Circus* hold in its prioritised version. What is more, **PRI** healthiness is guaranteed by CSP4 healthiness.

$$CSP4(P) =_{PSC} \mathbf{PRI}(CSP4(P)) = CSP4(\mathbf{PRI}(P))$$

Hiding

The problem with the semantics of hiding is that it doesn't satisfy one of the healthiness conditions - CSP3. While it was not a problem before, we have to be very careful with refusals when we add priority. For that reason we make sure that refusals of a previous process have no influence at the behaviour of hiding by unconstraining them inside the definition ($\exists s \bullet slots \cong s$).

$$A \setminus hidn \quad \widehat{=}_{PSC} \quad \mathbf{PRI} \circ \mathbf{R}3 \left(\begin{array}{l} \exists \mathbf{s}, s' \bullet A[\mathbf{s}, s'/slots, slots'] \wedge \\ \quad \mathbf{slots} \cong \mathbf{s} \wedge hidn \subseteq \bigcap srefs(s' \nwarrow \mathbf{s}) \\ \quad slots' \nwarrow slots = map(shide(hidn))(s' \nwarrow s) \wedge) \end{array} \right)$$

Timeout

The prioritised choice operator gives us an opportunity to improve one of the existing slotted-*Circus* operators. The original definition of timeout is based on the racing condition present implemented in the behaviour of external choice.

$$A \overset{d}{\rhd} B \quad \widehat{=}_{SC} \quad (A \Box (Wait\ d; int \to B)) \setminus \{ int \}$$

Unfortunately, the use of external choice introduces nondeterministic behaviour into timeout. That way the following property holds:

$$Skip \overset{0}{\rhd} B =_{SC} Skip \sqcap B$$

With prioritised choice we can define a deterministic and much more useful version of timeout.

$$[\text{Tout:pSC:def}] \qquad A \overset{d}{\rhd} B \mathrel{\widehat{=}}_{pSC} (A \overset{\rightarrow}{\Box} (Wait\ d; int \to B)) \setminus \{ int \}$$

That way we can formulate a law that precisely describes a timeout situation.

$$A \overset{0}{\rhd} B =_{pSC} B$$

The new definition of timeout also gives us an opportunity to formulate interesting algebraic laws, that are clearly explaining the details of racing conditions implemented in external and prioritised choice.

$$(1) \quad (a \to A \Box B) \setminus \{a\} =_{pSC} (A \setminus \{a\}) \sqcap (B \overset{1}{\rhd} Miracle) \setminus \{a\}$$
$$(2) \quad (A \overset{\leftarrow}{\Box} b \to 1B) \setminus \{b\} =_{pSC} (A \overset{1}{\rhd} B) \setminus \{ b \}$$

The first law holds because $Miracle$ is a unit of nondeterministic choice ($CSP1(P) \sqcap Mircale = CSP1(P)$). That way during the first clock cycle we get a nondeterministic choice between A and B, after which the low priority option B times out. The second law is based on the [SwapPri:law1]:p. 72. It states that if the low priority

option can perform an event in the first clock cycle, then the high priority option have only that one clock cycle to act, after which it will timeout. The usefulness of the given properties is still to be determined.

5.4 Prioritised CSP

After adding prioritised choice to occam there has been many papers trying to address problems with its implementation, and many attempts to formalise priority in CSP [Bar89] [Fid93]. Unfortunately, all the attempts resulted in the definition/implementation of priority that was unintuitive and always had some undesired properties. In here we try to prove that in fact defining an implementable and intuitive version of priority for CSP is impossible.

We start from two laws of CSP:

$$Stop \,\Box\, P =_{CSP} \quad P$$

$$Skip \,\Box\, P =_{CSP} \quad \begin{cases} Skip \sqcap P & \text{, if P terminates or performs an event} \\ Skip & \text{, otherwise} \end{cases}$$

If we now think about priority in a "Perfect PCSP" language and the two above laws, we get the two following desired properties:

$$(1) \quad Stop \overset{\leftarrow}{\Box} P =_{PerfectPCSP} P$$

$$(2) \quad Skip \overset{\leftarrow}{\Box} P =_{PerfectPCSP} Skip$$

$$(3) \quad DIV \overset{\leftarrow}{\Box} P =_{PerfectPCSP} P \text{ or } DIV$$

Prioritised choice is an implementation of external choice, so in case of (1) no other outcome is possible. However, in case of the second law we want priority to be deterministic version of external choice. Also the termination event is immediately available and beats any other behaviour that could be performed by the low priority option P. Finally the third law talks about prioritised choice between divergence and P. We do not fully define the solution other than it might be either DIV or P.

Because in CSP we can not measure the passing of time, the following law holds:

$$(4) \quad Q \setminus Events =_{CSP} \begin{cases} Skip, & \text{if Q terminates} \\ Stop \text{ or } DIV, & \text{otherwise} \end{cases}$$

If we now apply the desired laws of "Perfect PCSP" - (1), (2) and (3), to the law of CSP - (4), we get the following property:

$$(Q \setminus Events) \overset{\leftarrow}{\Box} P =_{PerfectPCSP} \begin{cases} Skip, & \text{if Q terminates} \\ P \text{ or } DIV, & \text{otherwise} \end{cases}$$

So, any implementation of "Perfect PCSP" would have to be able to solve the Halting problem.

Also because any CSP behaviour can be captured using slotted-*Circus* with the *CTA* trace model (inside a single slot), the above proof can be reduced to similar one in *CTA*. For that reason, *CTA* in a current, unrestricted form can not be used as a platform for priority.

5.5 Priority in the *CTA* History Model

In the previous section we argued that prioritised choice should not work with the *CTA* history model. At the other hand, as mentioned before (4.2.1), the *CTA* history model is equivalent to *MSA* when the timed communication is forced. The conclusion we can draw from here is that if anything is to go wrong in priority and *CTA*

then it will go wrong when we use untimed communication and immediate causality occurs.

$$a \rightarrow b \rightarrow Skip$$

If we now synchronise the process with prioritised choice and force the immediate causality with hiding we get the property below.

$$((a \rightarrow b \rightarrow Skip) \,||\, (b \rightarrow a \rightarrow Skip \overset{\leftarrow}{\Box} a \rightarrow b \rightarrow Skip)) \setminus \{a,b\} =_{CTA} Miracle$$

Because *CTA* records order in which events occur, only performing a followed by b can be accepted by the history model. Unfortunately once a is performed and b becomes available, prioritised choice decides that the high priority option is available and so falsifies the low priority one, in result turning the whole example into a $Miracle$.

Of course *MSA* also has unpredictable behaviour when immediate causality loops are present, but in case of the *MSA* history model, those cases are logical, can be explained and reasoned about. In case of *CTA* the interaction between priority and immediate causality can only be explained by not compatible notions of time and so is not worth reasoning.

When timed communication is enforced the *CTA* history model can be safely used with the current definition of prioritised slotted-*Circus*.

Chapter 6

Laws of Priority

In order to successfully use a new operator we need to be able to reason about any, even the most complicated cases when it is used. In this chapter we develop a set of laws that allow us to reason about prioritised choice. We do that in two ways. First by reducing a specification to its sequential form (a nondeterministic choice over a set of sequential programs). And second by introduction of a list of step laws which can be used where normalisation fails.

To simplify the laws and already complicated steps that take us to the normal form, we mostly do not mention channel based, input-output communication, and instead we treat input as external choice over possible events followed by assignment. While this assumption would be ineffective for the future tool support, it is good enough to prove the method being correct and allows for future improvements.

In this work we provide proofs of only a few core properties of the prioritised choice. The laws presented in this chapter are based on the understanding of these properties, and on the understanding of mechanisms defined in the denotational semantics of prioritised slotted-*Circus*. Unfortunately at the moment, because of the complexity of the semantics, lack of tool support and the amount of laws presented here, it is not feasible to provide formal proofs of the given laws.

6.1 Normalising

In this section we present steps which allow us to change a specification into its sequential form. Unfortunately normalisation steps often have a restriction to strong prefix healthiness or they need the specification to be complete (we can "unfold" a specification of any mentioned action). Also in this section we do not mention hiding. The laws of hiding are presented in the next section and should be used together with normalising steps whenever necessary. Finally, because preference is not symmetric and not transitive[1] the process of normalising and the laws included are complicated and no proofs of the normalising steps are given. In fact the methods used to prove basic laws of slotted-*Circus* can not be used to prove the normalising process because of their complexity. The question of how to do it is left unanswered. The normalisation process is made of two sets of steps that need to be iteratively applied to every parallel construct in the specification, starting from the bottom to the top of the specification tree. Every time the normalisation process is applied the parallel construct has a shorter chain of prefix operators to synchronise. This eventually leads to a situation where the parallel operator can be removed using one of the laws:

[Interleaving: unit law] $\quad Skip \mid\mid\mid A = A$

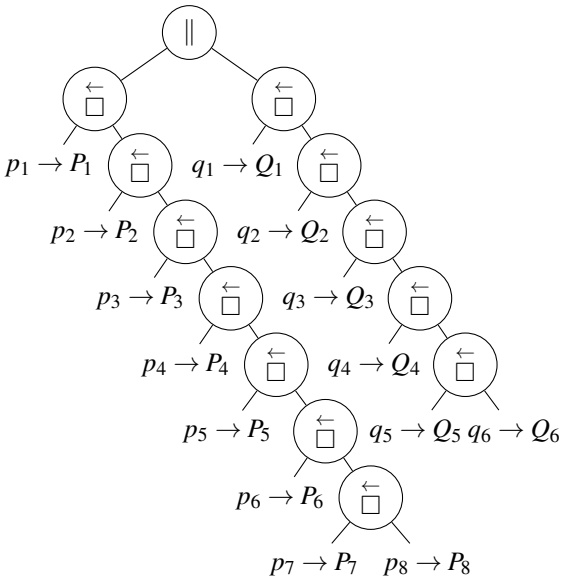
[1] Given two persons preferences - first one liking chocolate and apples but preferring apples, and the second disliking apples enough to prefer over them plain grass, we can not conclude that the two persons together would prefer to eat grass over chocolate.

$$[\text{Parallel: skip and comm}] \qquad Skip \underset{(H \cup \{a\})}{\parallel} a \rightarrow A = Stop$$

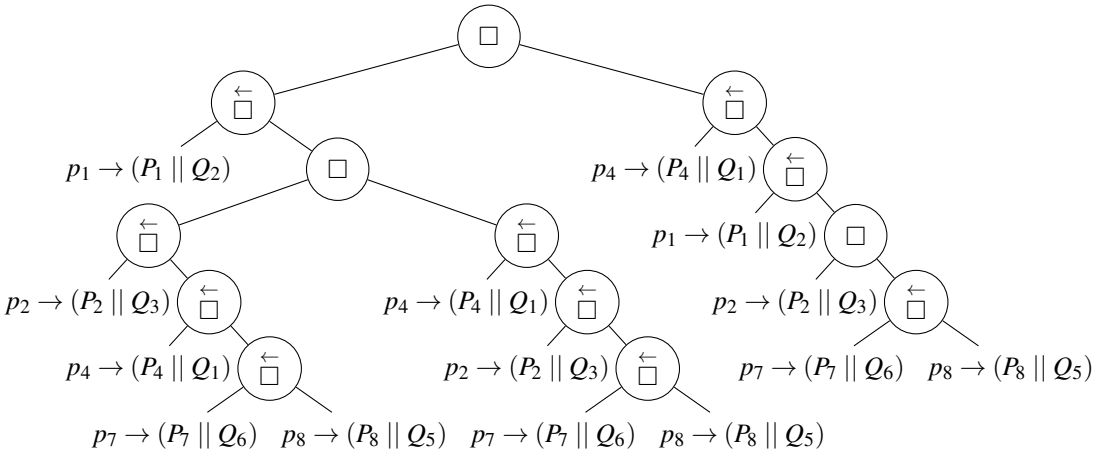
More complicated cases can be dealt with using the step laws presented later on or some of the normalisation steps.

The normalisation steps are explained using a graphical representation of a specified action. If we view a prioritised slotted-*Circus* specification as a tree, where nodes are operators and leaves are basic actions then the normalisation process is applied to the lowest parallel operators in the tree. The normalisation process is divided in to two stages. First allows us to pull operators such as conditional choice, nondeterministic choice and external choice above the closest parent parallel. Then it sorts the remaining priorities in a preference list. Because we talk about the lowest parallel construct we do not have to reason about interaction of parallel constructs. Also once we remove mentioned operators it is possible to sort the priorities (even though priority is non-transitive).

After the first normalisation process we end up with conditional, nondeterministic and external choices over the parallel compositions of two preference lists:



The second normalisation process is responsible for pushing the parallel construct further down, below priorities choices and the first communication events available. It is important to note that some of the laws from the second normalisation process only work properly with the strong healthy specifications. And so unsafe[2] specifications need to be processed with care.



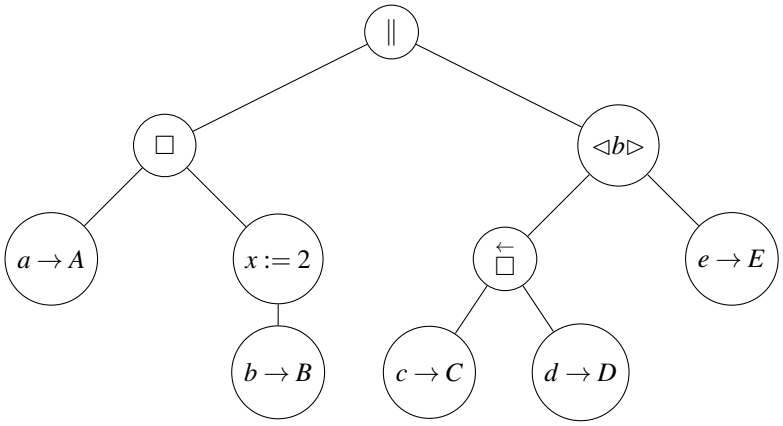
[2] By unsafe we mean a process that is not strong prefix healthy and so can introduce problems of cross communication and immediate causality

The two examples above are equivalent under following conditions: $p_1 = q_1, p_2 = p_5 = q_3 = q_4, p_4 = q_1, p_7 = q_6, p_8 = q_5$, and no cross communication.

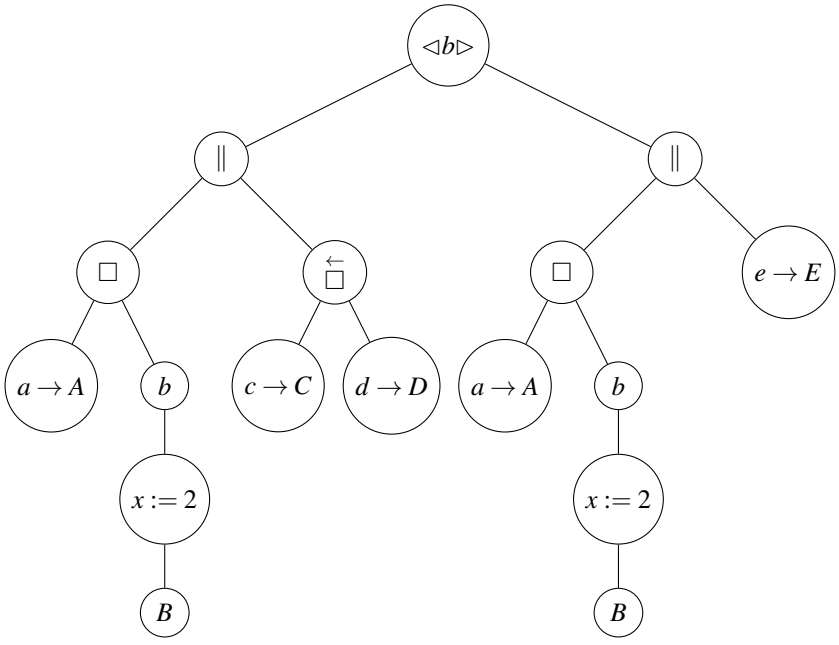
6.1.1 Creating priority lists

State operators

The first step in removing parallelism is dealing with state operators. Below we can see an example tree representing a part of an action we try to normalise. Because the parallel composition shown here is one of the bottom ones, we can assume in this and all the following steps that actions A, B, C, D, E do not mention parallelism.



In this step we are moving state operators (assignment, cond choice) behind communication, or above parallel composition.



Laws of assignment:

$$x := e; (A \,\Box\, B) = (x := e; A) \,\Box\, (x := e; B)$$
$$x := e; (A \sqcap B) = (x := e; A) \sqcap (x := e; B)$$
$$x := e; (A \overset{\leftrightarrow}{\Box} B) = (x := e; A) \overset{\leftrightarrow}{\Box} (x := e; B)$$
$$x := e; (A \lhd b \rhd B) = (x := e; A) \lhd b[e/x] \rhd (x := e; B)$$
$$x := e; c \to A = c[e/x] \to x := e; A$$

Laws of conditional choice operator:

$$(A \lhd b \rhd B) \,\Box\, C = (A \,\Box\, C) \lhd b \rhd (B \,\Box\, C)$$

[She06], 3.5.7, L.8 $$(A \lhd b \rhd B) \sqcap C = (A \sqcap C) \lhd b \rhd (B \sqcap C)$$

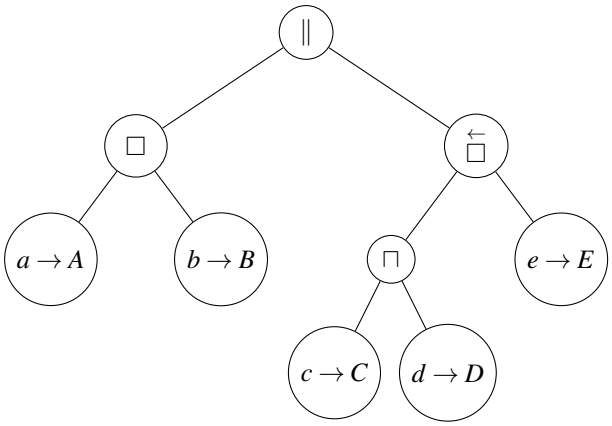
$$(A \lhd b \rhd B) \overset{\leftrightarrow}{\Box} C = (A \overset{\leftrightarrow}{\Box} C) \lhd b \rhd (B \overset{\leftrightarrow}{\Box} C)$$

[She06], 3.11, L.5 $$(A \lhd b \rhd B) \parallel C = (A \parallel C) \lhd b \rhd (B \parallel C)$$

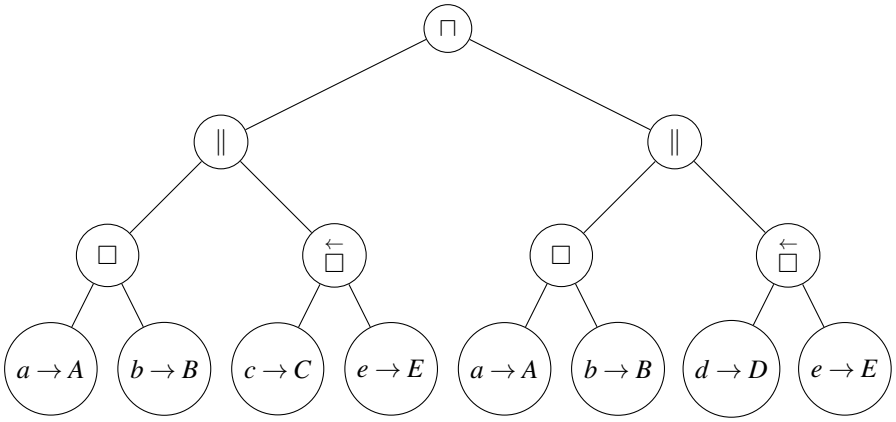
$$x := e; (A \lhd b \rhd B) = (x := e; A) \lhd b[e/x] \rhd (x := e; B)$$

Moving nondeterministic choice above parallel

In this step we work on a group of choices over communication followed by sequential actions.



Moving non-determinism higher then the lowest parallel operator in a tree.



Laws:

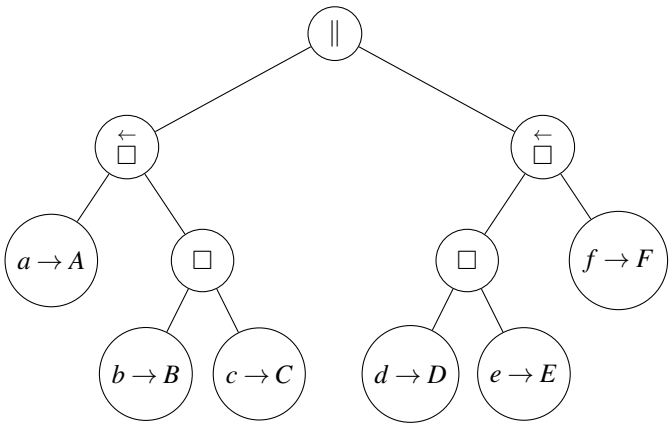
[Hoa85a], 3.3.1, L.7 $$(A \sqcap B) \,\Box\, C = (A \,\Box\, C) \sqcap (B \,\Box\, C)$$

$$(A \sqcap B) \overset{\leftrightarrow}{\Box} C = (A \overset{\leftrightarrow}{\Box} C) \sqcap (B \overset{\leftrightarrow}{\Box} C)$$

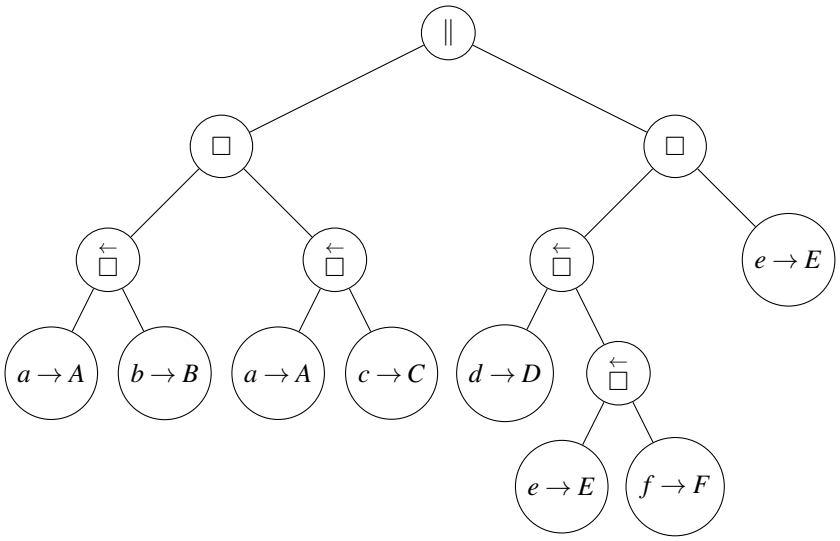
[Hoa85a], 3.2.1, L.7 $$(A \sqcap B) \parallel C = (A \parallel C) \sqcap (B \parallel C)$$

External choice in respect to prioritised choice

In this step we work on an external and prioritised choices over communication followed by sequential actions.



Moving external choice one step below the lowest parallel in the tree.

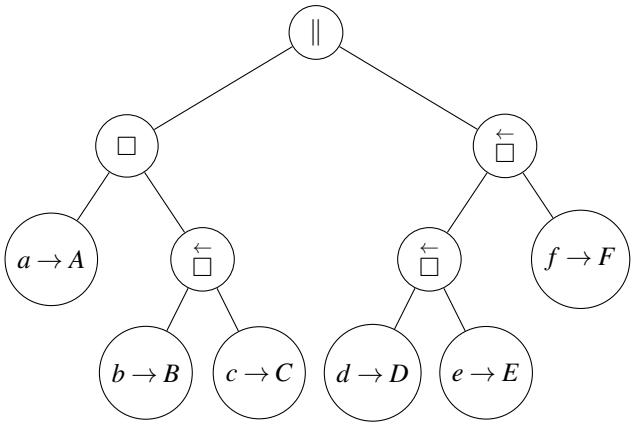


Laws:

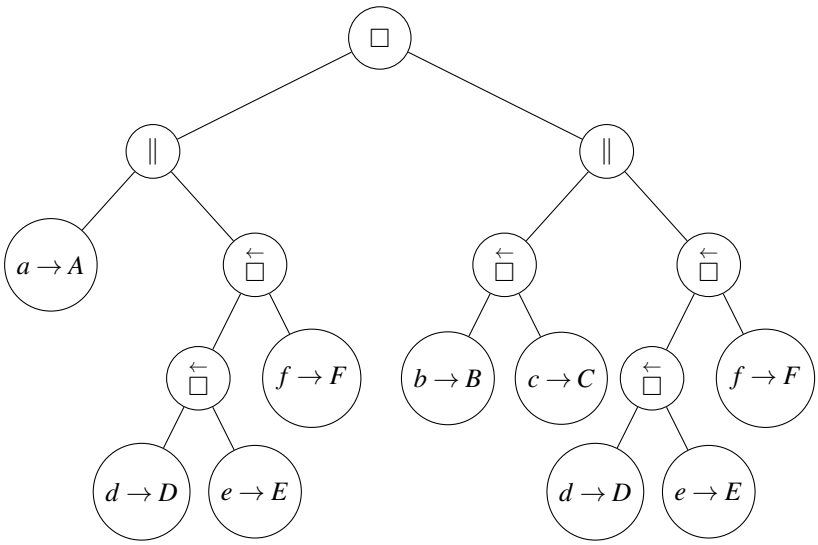
$$P \overleftarrow{\Box} (Q \Box S) = (P \overleftarrow{\Box} Q) \Box (P \overleftarrow{\Box} S)$$
$$(P \Box Q) \overleftarrow{\Box} S = (P \overleftarrow{\Box} Q \overleftarrow{\Box} S) \Box Q = (Q \overleftarrow{\Box} P \overleftarrow{\Box} S) \Box P$$

Moving external choice above parallel

In this step we work on an external choices over prioritised choices over communication followed by sequential actions.



Finally, after removing nondeterminism, we can handle the external choice



Laws:

$$[\text{EC:commutesPar}](A \,\Box\, B) \parallel C =_{SideCondition} (A \parallel C) \,\Box\, (B \parallel C)$$

Note that **this law only works for processes that are deterministic until first communication** (which at this stage of normalisation is guaranteed).

In other cases the following counterexample [3] holds:

$$(A \,\Box\, B) \parallel (C \sqcap D) = ((A \,\Box\, B) \parallel C) \sqcap ((A \,\Box\, B) \parallel D)$$

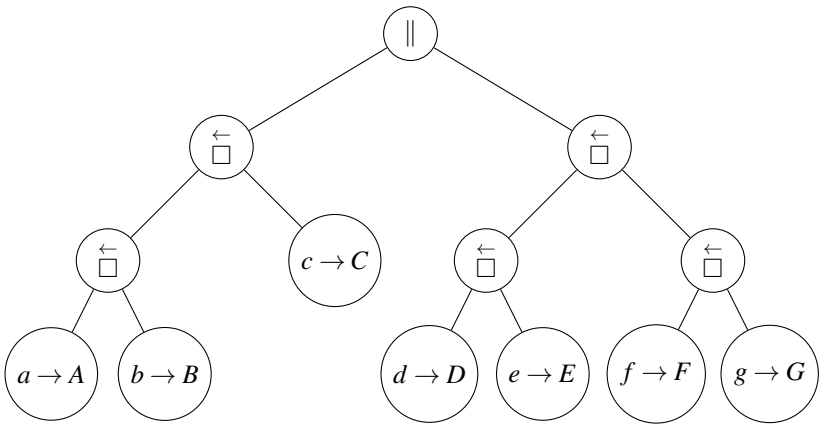
but

$$\begin{aligned}
&(A \parallel (C \sqcap D)) \,\Box\, (B \parallel (C \sqcap D)) \\
&= \\
&((A \parallel C) \sqcap (A \parallel D)) \\
&\Box\, ((B \parallel C) \sqcap (B \parallel D)) \\
&= \\
&((A \parallel C) \,\Box\, (B \parallel C)) \\
&\sqcap ((A \parallel C) \,\Box\, (B \parallel D)) \\
&\sqcap ((A \parallel D) \,\Box\, (B \parallel C)) \\
&\sqcap ((A \parallel D) \,\Box\, (B \parallel D))
\end{aligned}$$

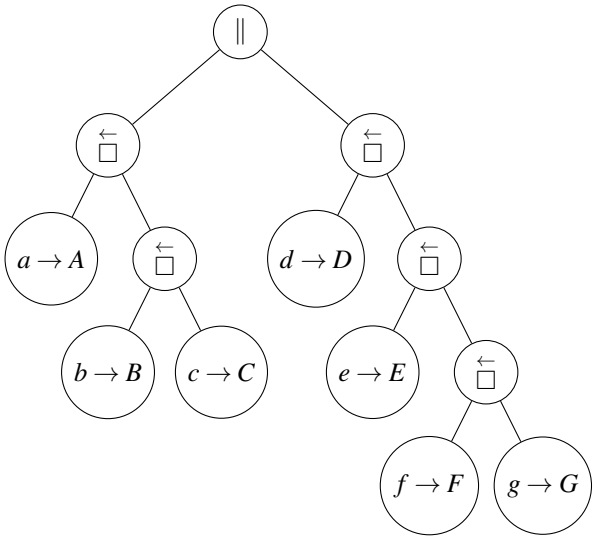
Stretching the preference tree

In this step we work on prioritised choices over communication.

[3] Counterexample was provided by Jim Woodcock, 18-10-2010



We can simplify the tree thanks to the connectivity laws of prioritised choice.



Laws:

$$(A \overleftarrow{\Box} B) \overleftarrow{\Box} C = A \overleftarrow{\Box} (B \overleftarrow{\Box} C)$$

6.1.2 Ordering priorities

Once we flatten the priority tree into a priority list we can represent the two parallel prioritised lists $P \parallel Q$ as two, finite strictly ordered sets of processes labelled with the first events they are ready to perform.

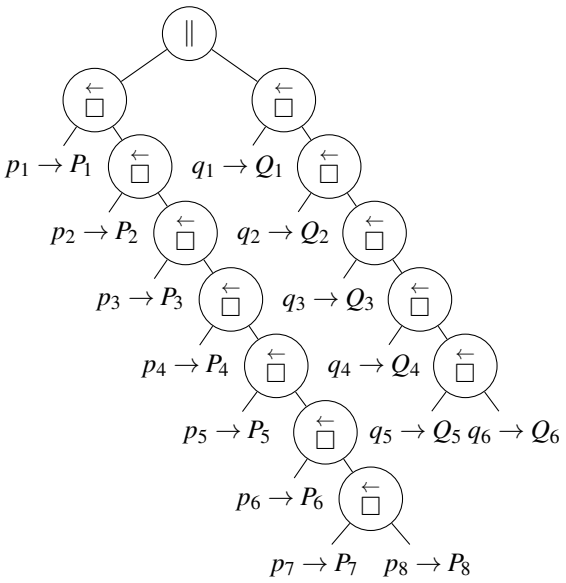
$$P \parallel Q$$

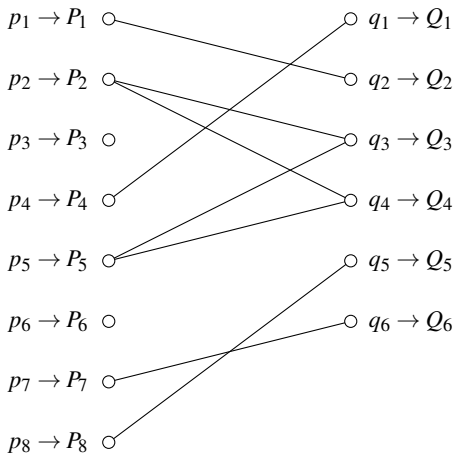
where

$$P = p_1 \to P_1 \overleftarrow{\Box} p_2 \to P_2 \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_n \to P_n$$

$$Q = q_1 \to Q_1 \overleftarrow{\Box} q_2 \to Q_2 \overleftarrow{\Box} \ldots \overleftarrow{\Box} q_m \to Q_m$$

At the right diagram we connect all parallel processes that want to perform the same event. In other words the right diagram is equivalent to the left if the following hold: $p_1 = q_2, p_2 = p_5 = q_3 = q_4, p_4 = q_1, p_7 = q_6, p_8 = q5$





Later in this section we will present the following notation to describe the normalisation step:

$$\begin{array}{rl} \text{IF} & \mathit{Condition} \\ \text{THEN} & P := \mathit{newP}(P) \end{array}$$

That way we will keep changing/simplifying (sometimes recursively) both P and Q. Also to note here is that in order to simplify the definitions of the normalising steps we assume that P and Q always have the following form:

$$P = p_1 \to P_1 \overleftarrow{\Box} p_2 \to P_2 \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_n \to P_n$$

$$Q = q_1 \to Q_1 \overleftarrow{\Box} q_2 \to Q_2 \overleftarrow{\Box} \ldots \overleftarrow{\Box} q_m \to Q_m$$

In other words even if we remove one of the options from P for example p_1:

$$P := p_2 \to P_2 \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_n \to P_n$$

Then the next step will still be defined on P ranging from p_1 to p_n.

Removing unnecessary nodes

WARNING this step will not work if causality loops are possible.[4]

If one of the actions wants to perform an event that can not be synchronised with any of the parallel options, then this action can be removed.

$$\begin{array}{rl} \text{IF} & (\exists i \in [1..n], \forall j \in [1..m] \bullet p_i \neq q_j) \\ \text{THEN} & P := (p_1 \to P_1 \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_{i-1} \to P_{i-1} \overleftarrow{\Box} p_{i+1} \to P_{i+1} \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_n \to P_n) \end{array}$$

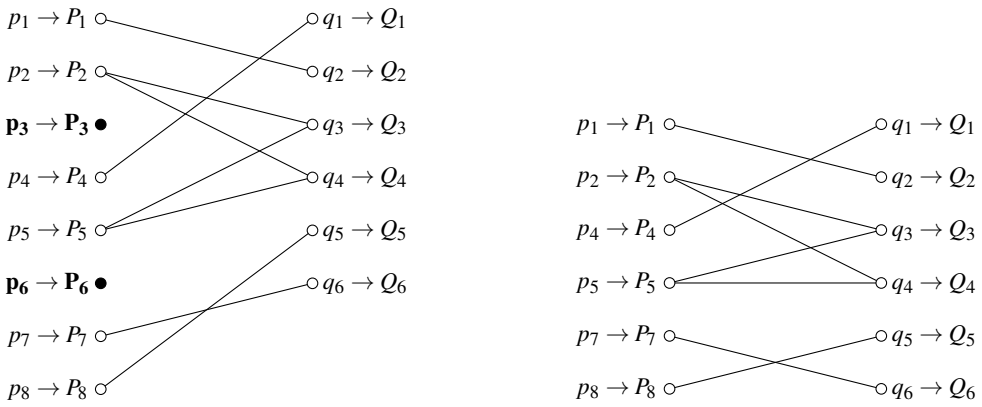
Symmetrically we simplify Q:

$$\begin{array}{rl} \text{IF} & (\exists j \in [1..m], \forall i \in [1..n] \bullet p_i \neq q_j) \\ \text{THEN} & Q := (q_1 \to Q_1 \overleftarrow{\Box} \ldots \overleftarrow{\Box} q_{j-1} \to Q_{j-1} \overleftarrow{\Box} q_{j+1} \to Q_{j+1} \overleftarrow{\Box} \ldots \overleftarrow{\Box} q_m \to Q_m) \end{array}$$

We perform this step until all the unnecessary nodes in P and Q are removed.

At the diagram the results can be observed as a removal of all the non connected actions.

[4] Counterexample: $a \to b \to P_1 \parallel ((b \to a \to Q_1) \overleftarrow{\Box} (a \to Q_2))$. Here Q_1 node can not be removed because there is a possibility of violating the prefix healthiness by synchronising $a \to b$ with $b \to a$.



For example:

$$(a \to A_1 \overleftarrow{\Box} b \to B_1 \overleftarrow{\Box} c \to C_1) \,||\, (a \to A_2 \overleftarrow{\Box} b \to B_2 \overleftarrow{\Box} d \to D_2)$$
$$= (a \to A_1 \overleftarrow{\Box} b \to B_1) \,||\, (a \to A_2 \overleftarrow{\Box} b \to B_2)$$

Removing repeating options

Based on law:

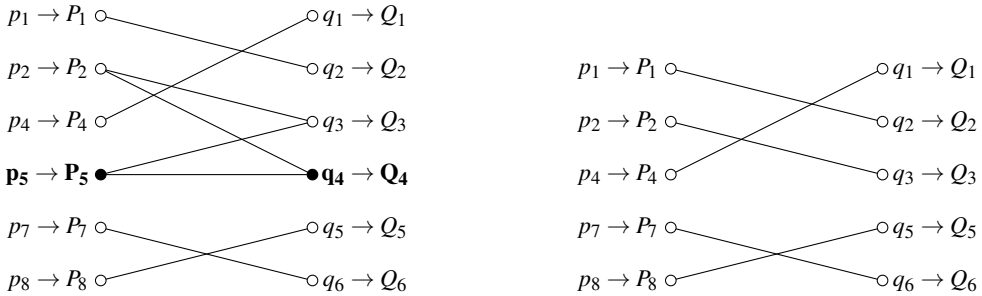
$$a \to P \overleftarrow{\Box} a \to Q = a \to P$$

We observe that if we have two processes wanting to perform the same event and one is of higher priority than the other, we can remove the low priority option as it can never be chosen. To address this property we can either use the law given above or perform the normalisation step presented below:

$$\begin{array}{ll} \text{IF} & (\exists i, j \in [1..n] \bullet i < j \wedge p_i = p_j) \\ \text{THEN} & P := (p_1 \to P_1 \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_{j-1} \to P_{j-1} \overleftarrow{\Box} p_{j+1} \to P_{j+1} \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_n \to P_n) \end{array}$$

Symmetric law should be applied to Q.

In the following diagram we can notice that now all actions are connected in one-to-one pairs.



For example:

$$(a \to P_1 \overleftarrow{\Box} a \to P_2) \,||\, Q = a \to P_1 \,||\, Q$$

Creating a priority tree (pushing the parallel construct down)

WARNING this step will not work if causality loops are involved.[5]

The last step of solving priorities is based on a list of properties concerning conflicting priorities:

$$(1) \qquad (A \overleftarrow{\Box} B) \,\Box\, (B \overleftarrow{\Box} A) = A \,\Box\, B$$

[5]Counterexample: $(a \to b \to P_1 \overleftarrow{\Box} b \to P_2) \,||\, ((b \to a \to Q_1) \overleftarrow{\Box} (a \to Q_2))$. Here the violation of the prefix healthiness is possible by synchronising $a \to b$ with $b \to a$.

external choice between conflicting priorities nullifies the preferences,

$$(2) \qquad (A \,\Box\, B) \overset{\leftarrow}{\Box} C \neq A \,\Box\, (B \overset{\leftarrow}{\Box} C)$$

external and prioritised choice are not connective (counter example: for B unavailable LHS reduces to $A \overset{\leftarrow}{\Box} C$ while RHS reduces to $A \,\Box\, C$),

$$(3) \qquad (A \,\Box\, B) \overset{\leftarrow}{\Box} C \neq (A \overset{\leftarrow}{\Box} C) \,\Box\, (B \overset{\leftarrow}{\Box} C)$$

prioritised choice does not distribute over external choice (counter example: for A unavailable LHS reduces to $B \overset{\leftarrow}{\Box} C$ while RHS reduces to $C \,\Box\, (B \overset{\leftarrow}{\Box} C) = C \,\Box\, B$),

$$(4) \qquad (A \,\Box\, B) \overset{\leftarrow}{\Box} C = (A \overset{\leftarrow}{\Box} B \overset{\leftarrow}{\Box} C) \,\Box\, (B \overset{\leftarrow}{\Box} A \overset{\leftarrow}{\Box} C)$$

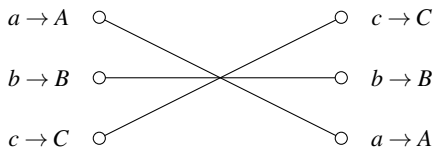
the final law replaces the standard distributivity law (3). As far as A and B are concerned it is equivalant to the law (1). In the case of C (in contrast to (3)) C is always in the lowest priority position below A and B - $(A \,\Box\, B) \overset{\leftarrow}{\Box} C$, $(A \overset{\leftarrow}{\Box} B) \overset{\leftarrow}{\Box} C$.

Even though those laws are not enough to fully explain or verify the properties below. They give an insight into rules of interaction between external and prioritised choice. First we have to define a set of top priority options (events that can be performed without any other, higher priority option restricting them). Those options can be recognised by the fact that there is no higher priority option on both of the sides of parallel composition.

$$TP \mathrel{\widehat{=}} \{(i,j) \bullet p_i = q_j \wedge \forall k,l \bullet (p_k \neq q_l \vee p_k < p_i \vee q_l < q_j)\}$$

For the example above TP (top priority set) is equal to $\{(1,2)(4,1)\}$ which represents $p_1 = q_2$ and $p_4 = q_1$. A thing to notice here is that $p_2 = q_3$ is not a highest priority option because $p_1 = q_2$ will be of higher priority. A very important thing to note here is that there can be more than two top priority options. For example:

$$(a \to A \overset{\leftarrow}{\Box} b \to B \overset{\leftarrow}{\Box} c \to C) \,||\, (c \to C \overset{\leftarrow}{\Box} b \to B \overset{\leftarrow}{\Box} a \to A)$$



will have three top priority options and will be equal to the external choice between those options.

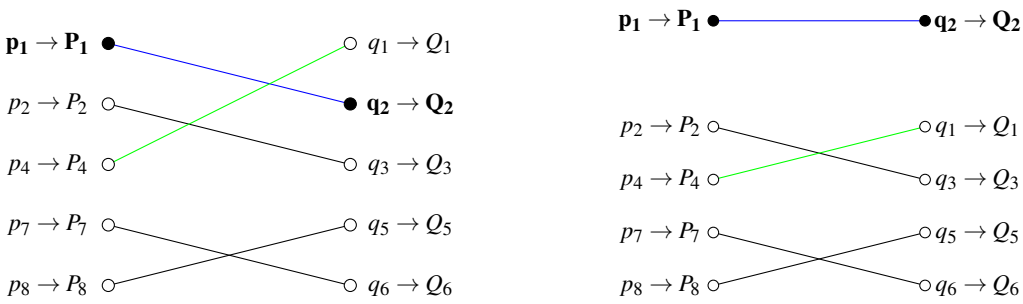
Once we have a set of top priority options we can perform an inductive step reducing the number of options in parallel composition.

$$\begin{aligned} P \,||\, Q \;:=\; & \Box_{(i,j)\in TP}(p_i \to (P_i \,\|\, Q_j)) \overset{\leftarrow}{\Box} \\ & ((p_1 \to P_1 \overset{\leftarrow}{\Box} \ldots \overset{\leftarrow}{\Box} p_{i-1} \to P_{i-1} \overset{\leftarrow}{\Box} p_{i+1} \to P_{i+1} \overset{\leftarrow}{\Box} \ldots \overset{\leftarrow}{\Box} p_n \to P_n) \\ & \| \, (q_1 \to Q_1 \overset{\leftarrow}{\Box} \ldots \overset{\leftarrow}{\Box} q_{j-1} \to Q_{j-1} \overset{\leftarrow}{\Box} q_{j+1} \to Q_{j+1} \overset{\leftarrow}{\Box} \ldots \overset{\leftarrow}{\Box} q_n \to Q_n)) \end{aligned}$$

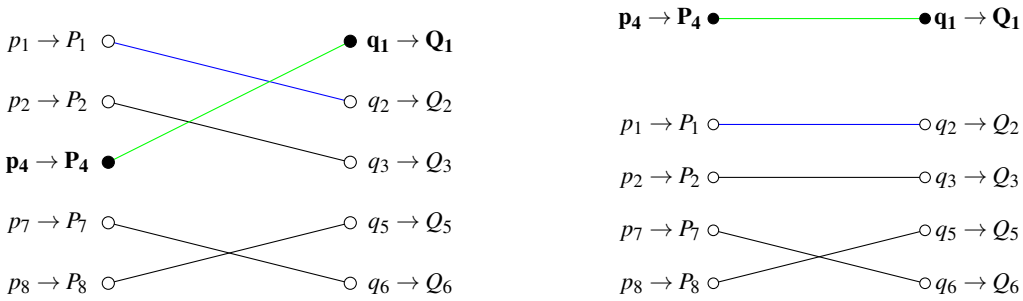
If there is only one top priority option then the above law can be reformulated into:

$$(a \to A_1 \overset{\leftarrow}{\Box} B_1) \,||\, (a \to A_2 \overset{\leftarrow}{\Box} B_2) = a \to (A_1 \,||\, A_2) \overset{\leftarrow}{\Box} (B_1 \,||\, B_2)$$

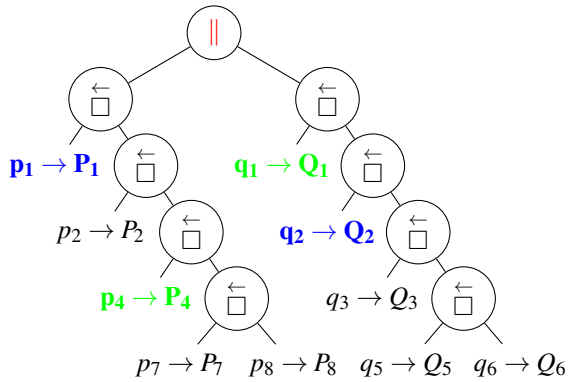
In our example we can present the above step by drawing diagrams for each of the top priority pair:
$(\mathbf{1},\mathbf{2}) \in \mathbf{TP}$



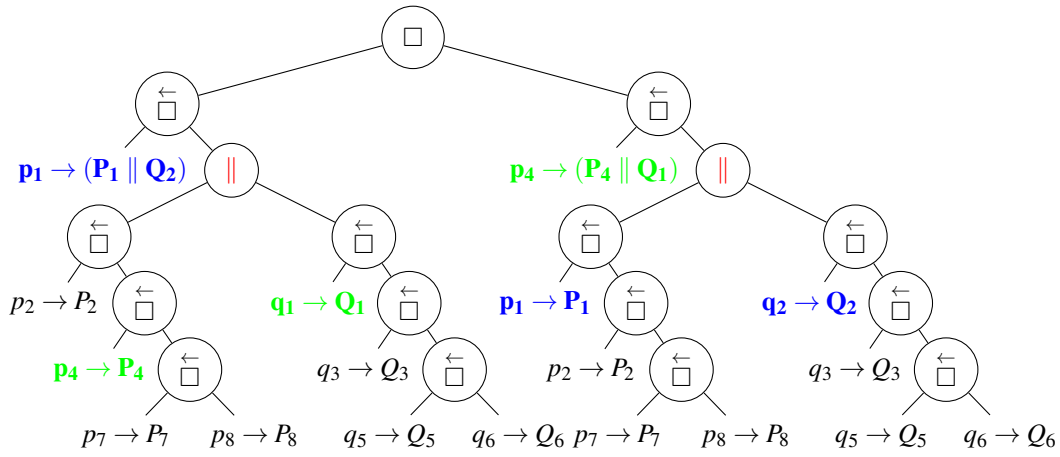
$(\mathbf{4},\mathbf{1}) \in \mathbf{TP}$



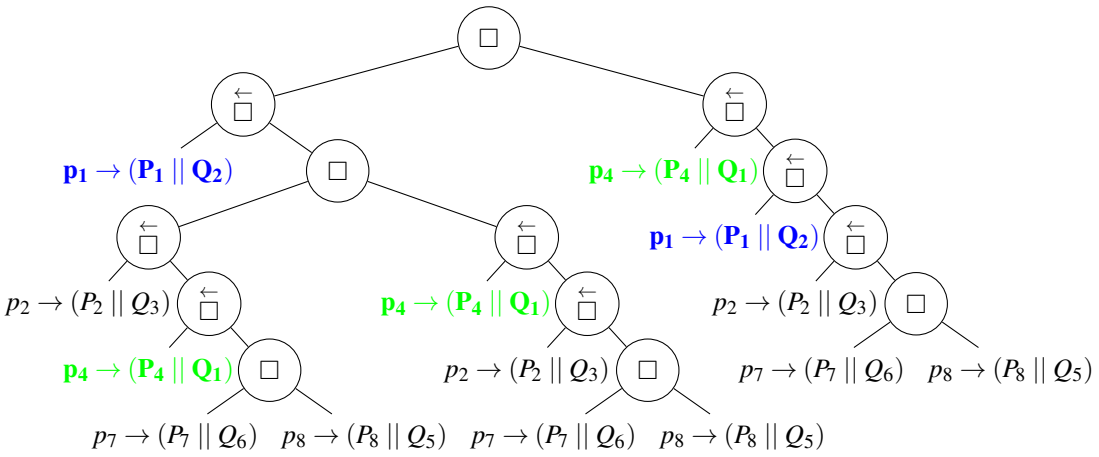
On the tree diagram we can see how the above step transforms one parallel composition of two priority lists:



into two, similar but one level smaller parallel compositions (here the left branch represents $(1,2)$, and the right one $(4,1)$ top priority pair),



We can further iterate this normalisation step and end up pushing the parallel operator behind any communication. That way we end up with external and prioritised choices between prefixes, so the given example will reduce into the following tree:



At this point we can start the normalisation process from the beginning for every leaf of the given tree.

6.2 Hiding

When describing the normalising steps we haven't mentioned the hiding operator that might occur at any normalising steps.

The most important law of hiding is when it ends the scope of an event and enforce its execution.

$$[\text{Hide:law1}] \qquad (a \to A)\setminus(H \cup \{a\}) = A\setminus(H \cup \{a\})$$

At the same time hiding is ignoring events that it does not mention.

$$[\text{Hide:law2}] \qquad (a \to A)\setminus(H \setminus \{a\}) = a \to A\setminus(H \setminus \{a\})$$

Other simple laws that might be useful (not only) during the normalisation process are listed below.

Laws:

$$[\text{She06}], 3.14, L.11 \quad A;B\setminus H = A\setminus H;B\setminus H$$

$$[\text{Hoa85a}], 3.5.1, L.3 \quad A \sqcap B\setminus H = A\setminus H \sqcap B\setminus H$$

$$[\text{Hoa85a}], 3.5.1, L.5 \quad (a \to Skip)\setminus(H \cup \{a\}) = Skip$$

$$(a \to A \overleftarrow{\Box} B)\setminus(H \cup \{a\}) = A\setminus(H \cup \{a\})$$

$$(a \to A \overleftarrow{\Box} b \to B \overleftarrow{\Box} c \to C)\setminus(H \cup \{b\}) = (a \to A \overleftarrow{\Box} b \to B)\setminus(H \cup \{b\})$$

$$(a \to A \overleftarrow{\Box} b \to B)\setminus H = (a \to A\setminus H \overleftarrow{\Box} b \to B\setminus H)\setminus H$$

$$(a \to A \,\Box\, b \to B)\setminus H = (a \to A\setminus H \,\Box\, b \to B\setminus H)\setminus H$$

$$[\text{Hoa85a}], 3.5.1, L.1 \quad (A\setminus H_1)\setminus H_2 = A\setminus(H_1 \cup H_2)$$

6.2.1 Redirecting priorities

An interesting property of prioritised choice, that gives insight to the timing details of preference, is the ability to swap priorities.

$$[\text{SwapPri:law1}] \quad (A \overleftarrow{\Box} b \to wait\ 1;B)\setminus H \cup \{b\} = (A \overrightarrow{\Box} wait\ 1;b \to B)\setminus H \cup \{b\}$$

The low priority option of the LHS $b \xrightarrow{1} B$ is willing to perform an event immediately. That restricts the high priority option A to perform an event in the firs clock cycle or to be timed-out. On the RHS A is a low priority option, but still can only perform an event during the first clock cycle, else it will be timed out by the high priority option $wait\ 1;b \to B$ that will perform an event in the second clock cycle.

While $b \rightarrow wait\ 1; B \neq wait\ 1; b \rightarrow B$, because of the availability of event b in the first clock cycle, the above law still holds because b is an internal event and can not be observed.

$$(b \rightarrow wait\ 1; B) \setminus \{b\} = (wait\ 1; b \rightarrow B) \setminus \{b\} = wait\ 1; B \setminus \{b\}$$

An equivalent law that comes in handy when dealing with the timeout definition can be formulated by addition of an internal event int.

$$[\text{SwapPri:law2}] \quad (A \overset{\leftarrow}{\Box} b \xrightarrow{1}; B) \setminus H \cup \{int, b\} = (A \overset{\rightarrow}{\Box} wait\ 1; int \rightarrow B) \setminus H \cup \{int, b\}$$

6.3 Step Laws

While normalising steps are the easiest way of dealing with complicated specifications, there are some very strong restrictions present on them. Almost every law presented in the "Ordering priorities" sub-section 6.1.2 is restricted to actions that are strong prefix healthy. Normalising steps are also designed to work on the lowest parallel constructs in the specification tree. This means that not knowing the behaviour of the whole specification (recursion or undefined processes might be present in the specification) may ensure that finishing the normalisation process becomes impossible. A good example of a step that might be impossible to follow is [EC:commutesPar]:p. 66. For the given reasons during the work on prioritised slotted-*Circus* a number of step laws have been developed and used when dealing with specifications.

6.3.1 Step Law 1

Two parallel processes ($a \rightarrow A$ and $b \rightarrow B$) are willing to perform different events ($a \neq b$).

$$(a \rightarrow A \underset{\{a\}}{\parallel} b \rightarrow B)$$

Because they are synchronised only on an event offered by the left process (a), which is blocked by the right process not agreeing to perform it, the event b can be performed first.

$$(a \rightarrow A \underset{\{a\}}{\parallel} b \rightarrow B) = b \rightarrow (a \rightarrow A \underset{\{a\}}{\parallel} B)$$

The same rule can be applied if more than one event is offered by the both sides assuming that $\forall i \in \langle 1..n \rangle, j \in \langle 1..m \rangle \bullet a_i \neq b_j$.

$$[\text{StepLaw1}] \quad \begin{array}{l} (\underset{i \in \langle 1..n \rangle}{\Box} (a_i \rightarrow A_i)) \underset{H \cup \{b_1..b_m\} \setminus \{a_1..a_n\}}{\parallel} (\underset{j \in \langle 1..m \rangle}{\Box} (b_j \rightarrow B_j)) \\ = \\ \underset{i \in \langle 1..n \rangle}{\Box} \left(a_i \rightarrow \left(A_i \underset{H \cup \{b_1..b_m\} \setminus \{a_1..a_n\}}{\parallel} (\underset{j \in \langle 1..m \rangle}{\Box} (b_j \rightarrow B_j)) \right) \right) \end{array}$$

The law also holds for prioritised and nondeterministic choice

$$\begin{array}{l} (\underset{i \in \langle 1..n \rangle}{\overset{\leftarrow}{\Box}} (a_i \rightarrow A_i)) \underset{H \cup \{b_1..b_m\} \setminus \{a_1..a_n\}}{\parallel} (\underset{j \in \langle 1..m \rangle}{\Box} (b_j \rightarrow B_j)) \\ = \\ \underset{i \in \langle 1..n \rangle}{\overset{\leftarrow}{\Box}} \left(a_i \rightarrow \left(A_i \underset{H \cup \{b_1..b_m\} \setminus \{a_1..a_n\}}{\parallel} (\underset{j \in \langle 1..m \rangle}{\Box} (b_j \rightarrow B_j)) \right) \right) \end{array}$$

6.3.2 Step Law 2

If two parallel processes can perform only one, the same event then they will synchronise on it. This law describes one of the main differences between the CSP and CCS based theories.

$$[\text{StepLaw2}] \qquad (a!x \to A) \underset{H \cup \{|a|\}}{||} (a?x \to B) = (a!x \to (A \underset{H \cup \{|a|\}}{||} B)$$

$$[\text{StepLaw2b}] \qquad (a \to A) \underset{H \cup \{a\}}{||} (a \to B) = (a \to (A \underset{H \cup \{a\}}{||} B)$$

6.3.3 Step Law 3

This law represents the ability of prioritised choice to enforce the preference on interleaved processes. Because it might not hold if cross communication is performed [6] it should assume the strong prefix healthiness or be used with care when untimed communication is used.

$$[\text{StepLaw3}] \qquad \left(((a \to A_1) \,|||\, (b \to B_1)) \underset{H \cup \{a\}}{||} ((a \to A_2) \overset{\leftarrow}{\Box} (b \to B_2)) \right) \setminus (H \cup \{a\}) = \left((A_1 \,|||\, (b \to B_1)) \underset{H \cup \{a\}}{||} A_2 \right) \setminus (H \cup \{a\})$$

This law is a very important property of prioritised choice because in fact this is the only way in which priority can influence the behaviour of the environment. Similar laws where external or nondeterministic choice is used instead of interleaving do not hold.

$$\left(((a \to A_1) \,\Box\, (b \to B_1)) \underset{H \cup \{a\}}{||} ((a \to A_2) \overset{\leftarrow}{\Box} (b \to B_2)) \right) \setminus (H \cup \{a\}) \neq \left(A_1 \underset{H \cup \{a\}}{||} A_2 \right) \setminus (H \cup \{a\})$$

The main reason for this is because the nondeterminism of external, or nondeterministic choice, is stronger than any preference. External choice can originate from conflicting priorities ($A \,\Box\, B = (A \overset{\leftarrow}{\Box} B) \,\Box\, (A \overset{\rightarrow}{\Box} B)$) and because no weight are assigned to them, external choice will always remove any preference. Formally a counterexample can be shown based on the [EC:commutesPar]:p. 66 law. For example:

[6] $\left(((a \to A_1) \,|||\, (b \to a \to B_1)) \underset{H \cup \{a\}}{||} ((a \to b \to A_2) \overset{\leftarrow}{\Box} (b \to B_2)) \right) \setminus (H \cup \{a\})$ In this case the high priority option $a \to b \to a_2$ can synchronise with $b \to a \to B_1$, by performing cross communication.

$$\left(((a \to c \to Skip) \,\Box\, (b \to d \to Skip)) \underset{\{a,b\}}{||} ((a \to Skip) \overset{\leftarrow}{\Box} (b \to Skip)) \right) \setminus \{a,b\}$$

$\equiv$ " [EC:commutesPar]:p. 66 "

$$\left(\begin{array}{l} ((a \to c \to Skip) \underset{\{a,b\}}{||} ((a \to Skip) \overset{\leftarrow}{\Box} (b \to Skip))) \\ \Box\, ((b \to d \to Skip) \underset{\{a,b\}}{||} ((a \to Skip) \overset{\leftarrow}{\Box} (b \to Skip))) \end{array} \right) \setminus \{a,b\}$$

$\equiv$ " removing unnecessary options (6.1.2) "

$$\left(\begin{array}{l} ((a \to c \to Skip) \underset{\{a,b\}}{||} (a \to Skip)) \\ \Box\, ((b \to d \to Skip) \underset{\{a,b\}}{||} (b \to Skip)) \end{array} \right) \setminus \{a,b\}$$

$\equiv$ " [StepLaw2b]:p. 74 "

$$\left(a \to ((c \to Skip) \underset{\{a,b\}}{||} Skip) \,\Box\, b \to ((d \to Skip) \underset{\{a,b\}}{||} Skip) \right) \setminus \{a,b\}$$

$\equiv$ " [StepLaw1]:p. 73 "

$$\left(a \to c \to Skip \,\Box\, b \to d \to Skip \right) \setminus \{a,b\}$$

$\equiv$ " property of external choice "

$$c \to Skip \sqcap d \to Skip$$

6.3.4 Step Law 4

Because only one event of the events offered by interleaved processes is able to synchronise. We can synchronise and pull it in front of the parallel operator.

$$[\text{StepLaw4}] \qquad ((a \to A_1) \,|||\, (b \to B_1)) \underset{H \cup \{a\}}{||} (a \to A_2)$$
$$= a \to \left((A_1 \,|||\, (b \to B_1)) \underset{H \cup \{a\}}{||} A_2 \right)$$

here $a \neq b$.

Chapter 7

Case Studies

7.1 Handel-C

7.1.1 Intro

Handel-C was chosen as a target language for slotted-*Circus* because of its precise notion of time and because its communication model is based on CSP and as follows, similar to slotted-*Circus*. In order to prove that slotted-*Circus* is expressive and fits the purpose as a specification language for real time synchronous hardware, it was decided that not only code generation from slotted-*Circus* to implementation language should be simple but we should be able to model the behaviour of the hardware itself. We want slotted-*Circus* to be as close to hardware language as possible without losing the mathematical properties.

The Handel-C operator that proved to be very difficult to formally describe was `prialt` which offers a prioritised choice between available cases.

```
prialt
{
       case a ?  x:
            x++;
            break;
       case b !  y:
            y++;
            break;
}
...
```

The presented code awaits for input on channel `a` and output on channel `b`. If none of the communications are available then the prialt construct will allow time to pass. If only one case is available within a clock cycle then `prialt` will execute it. If both of the cases are available within the same clock cycle then `prialt` will execute the higher priority option (`a ? x: x++;`) which will read a value on the input channel to the variable, perform a clock tick (communication takes time), increment the variable, perform a clock tick (assignment takes time) and terminate.

If we compare the description of the `prialt` operator with the description of prioritised choice we will notice that they are in fact the same. That way we can directly translate the given example into a prioritised slotted-*Circus* process.

$$(a?x \xrightarrow{1} x := x+1; Wait\ 1) \\ \overleftarrow{\Box}\ (b!y \xrightarrow{1} y := y+1; Wait\ 1)$$

Even more complicated to formally model was the `prialt` statement with the `default` case.

```
prialt
{
        case a ?  x:
             x++;
             break;
        case b !  y:
             y++;
             break;
        default:
             z++;
             break;
}
...
```

While without `default` the execution of `prialt` halts until one of the cases is ready to communicate, the presence of `default` means that if no communication is available immediately then the `default` branch will be performed instead. It is important to note here that the default branch will be performed without any delays (opposite to the standard cases which always have to reserve one clock cycle for communication).

Happily prioritised slotted-*Circus* can model this behaviour with ease and below we present two possible solutions.

$$\left(\begin{array}{l} a?x \xrightarrow{1} x := x+1; Wait\ 1; \\ \overleftarrow{\Box}\ b!y \xrightarrow{1} y := y+1; Wait\ 1; \\ \overleftarrow{\Box}\ default \rightarrow z := z+1; Wait\ 1; \end{array} \right) \setminus default$$

The first one is using a special event $default$ that triggers the lowest priority default option immediately and not allowing the higher priority options to wait.

$$def := false; \\ \left(\begin{array}{l} a?x \xrightarrow{1} x := x+1; Wait\ 1; \\ \overleftarrow{\Box}\ b!y \xrightarrow{1} y := y+1; Wait\ 1; \\ \overleftarrow{\Box}\ def := True; \end{array} \right); \\ (z := z+1; Wait\ 1) \lhd def \rhd Skip$$

The second is a little more complex but in return is not using the undesired untimed communication. Instead of triggering the default clause with the $default$ event, this time we use an immediate termination. The termination is recorded in state with the boolean variable def, and the body of the default case is performed immediately after.

The only difference between the two given examples lies in either introducing a special control event "default" or a control variable "def". All the other details (timing, observable events, state excluding def) of behaviour of the two actions are the same.

7.1.2 The Prialt Trick

While being very problematic untimed communication is often used in hardware implementations. In Handel-C untimed communication is possible by the use of a special variable type called signal. Signals should be thought of as physical wires or asynchronous communication, rather than the standard CSP communication or a variable. The communication via signal takes no time and allows a passage of a data from parallel parts of code within a clock cycle. Also the assigned value of the signal is only visible until a clock-tick occurs and if no value is assigned to it during a clock cycle then for the duration of the clock cycle it assumes a predefined default. The danger of this solution is similar to the use of untimed communication in slotted-*Circus*. For that reason while it is possible to describe oscillating hardware loops in Handel-C they are always detected by the compiler.

While searching for an interesting case study we came upon what's called the prialt trick [Mat07]. The prialt trick makes use of details of timing properties of many Handel-C operators including parallelism, `prialt` with the default clause and the signal variables. For that reason it is very well suited to the target of proving the expressiveness of prioritised slotted-*Circus*.

Modelling the behaviour of signals with timed prioritised slotted-*Circus* is impossible, but it is relatively simple when no restrictions of the timed communication are enforced. Assignment to a signal variable is modelled as an asynchronous broadcast:

$$SigName!_{sig}x \mathrel{\widehat{=}} (SigName!x \rightarrow Skip) \overleftarrow{\Box} Skip$$

while whenever we try to read the variable we perform an asynchronous input that is recorded in a *temp* variable.

$$SigName?_{sig}temp \mathrel{\widehat{=}} (SigName?temp \rightarrow Skip) \overleftarrow{\Box} (temp := SigNameDefault)$$

Note that both of the communications are always performed within a clock cycle similar to operations on the signal variable. Also if we try to read a value of the signal that was not broadcasted, we assign a default value to the result.

"Prialt Trick" Handel-C code [Mat07]

```
macro proc PrialtTrickReceive (ChannelPtr, ActionMacro) {
    signal unsigned 1 Rx = 1;
    signal Data;
    while (1) {
        par {
            prialt {
                case *ChannelPtr ?  Data: break;
                default:  Rx = 0; break;
            }
            if (Rx == 1) {
                ActionMacro (Data);
            }
            else {
                delay;
            }
        }
    }
}
```

The "prialt trick" is able to react in parallel to incoming messages at the same clock cycle as they are being send. That allows the processing of the incomming data without any delay (within a clock cycle), which would

normally be necessary for recording the data in the memory. In order to understand the implementation of the "prialt trick" we first have to explain the timing details of the Handel-C operators. All assignments and channel communication events take a clock cycle. All conditionals and expressions evaluations along with priority resolutions are instantaneous (completed before the end of current clock cycle). It is possible to read a communicated value within a clock cycle if we use a signal variable. The communication will still wait till the end of the clock cycle (blocking all the sequentially composed processes), but the signal variable can be processed by a parallel process. As for the prialt operator the choice is made only when one of the available cases performs an event and so it takes clock cycle. An exception is the default clause that is not blocked by communication and so it can be performed instantaneously. This is possible because in the hardware level the default clause is implemented as a simple wire, while communication involves a clocked register.. When any of the other cases becomes available the default clause is interrupted. Because assignment takes a full clock cycle (the values of variables are only changed at the end of the clock cycle when hardware becomes stable) no rolling back is necessary.

The corresponding behaviour can be described using prioritised slotted-*Circus*. In here as described before we use untimed communication for describing the behaviour of signals as described before, where thanks to prioritised choice we can encapsulate the fact that the value of signals is temporary and can only be read within the same clock cycle as it has been assigned. **Slotted-Circus code:**

$$
\begin{array}{l}
RxDefault := 1; DataDefault :=?; \\
while(True) \\
(\\
\quad \left(\begin{array}{l} \left(ChanelPtr?x_1 \rightarrow Data!_{sig}x_1; Wait1 \right) \\ \overleftarrow{\Box} \\ (Rx!_{sig}0) \end{array} \right) \\
\quad | \{Rx, Data\} | \\
\quad \left(Rx?_{sig}x_2; (Data?_{sig}x_3; ActionMacro(x_3)) \lhd x_2 == 1 \rhd (Wait\ 1) \right) \\
) \setminus \{Rx, Data\}
\end{array}
$$

Where the signal communication is defined as:

$$SigName!_{sig}x \mathrel{\widehat{=}} (SigName!x \rightarrow Skip) \overleftarrow{\Box} Skip$$

$$SigName?_{sig}temp \mathrel{\widehat{=}} (SigName?temp \rightarrow Skip) \overleftarrow{\Box} (temp := SigNameDefault)$$

.

The *while* loop used above is not a part of the slotted-*Circus* syntax but can be easily defined using the available recursion: $while(True)(P) = \mu X \bullet P; X$.

An important aspect of the prioritised slotted-*Circus* specification is that it is unsafe for the same reasons as the Handel-C program. We can imagine a situation when *ActionMacro* tries to send something on *ChannelPtr*. This will create a hardware loop and will force Handel-C compiler to return an error.

The conclusion from this example is that prioritised slotted-*Circus* without any restrictions on immediate-causality is expressive enough to be used as an implementation language, but the consequence of that is the need for compile time analysis of the code in search for oscillating loops.

7.2 WSN

The difficulty in the modelling of the behaviour of wireless sensor networks lies in the complexity and uniqueness of the communication model that they are based on. Nodes in WSN-s are often unreliable and the topology

is (at a different level) dynamic. Any outgoing communication is in the form of a broadcast and collisions are a common problem. While collision detection on the senders side is not an issue (sender listens on the channel just before broadcasting, reducing the probability of collisions to acceptable minimum), dealing with it on the receivers side is often very complicated. A receiving node can be within a range of two broadcasting nodes that transmit at the same time but are too far from each other to detect the collision themselves. When this happens only nodes that are within the overlapping range experience colliding transmission and the rest of the network works correctly. This kind of behaviour is very difficult to model using the standard slotted-*Circus* because of lack of asynchronous communication and difficulty in modelling collisions.

Happily prioritised slotted-*Circus* can address the special features of WSN-s with ease. A broadcasting node, in contrast to the CSP model of communication, never waits for the environment to be ready for its transmission. If for some reason the topology of a network will change, or there is a synchronisation issue with one of the receivers, the broadcasting node will still send its data and will not wait. In the most extreme situation there will be no nodes listening to senders transmissions, but this should not block the senders operations. One of the ways we can model the nonblocking communication in slotted-*Circus* is:

$$Send(ch,e) \mathrel{\widehat{=}}_{SC} Wait\ 1 \Box (ch!e \xrightarrow{1} Skip)$$

Unfortunately, because of the nondeterminism present in the external choice, the behaviour of this process in the second clock cycle is nondeterministic (it is possible that a process will synchronise with it after the first clock tick), which is different from the situation that we are trying to describe. For example [1]:

$$\begin{array}{ll} & \left(\ Send(ch,e) \ ||\ (Wait\ 1; ch?x \xrightarrow{1} Skip) \ \right) \setminus \{ch\} \\ \equiv & \text{“ } Send \text{ definition without priority ”} \\ & \left(\ (Wait\ 1 \Box (ch!e \xrightarrow{1} Skip)) \ ||\ (Wait\ 1; ch?x \xrightarrow{1} Skip) \ \right) \setminus \{ch\} \\ \equiv & \text{“ Synchronising on waiting ”} \\ & Wait\ 1; \left(\ (Skip \Box (ch!e \xrightarrow{1} Skip)) \ ||\ ch?x \xrightarrow{1} Skip \ \right) \setminus \{ch\} \\ \equiv & \text{“ Property of external choice ”} \\ & Wait\ 1; \left(\ (Skip \ ||\ ch?x \xrightarrow{1} Skip) \sqcap (ch!e \xrightarrow{1} Skip \ ||\ ch?x \xrightarrow{1} Skip) \ \right) \setminus \{ch\} \\ \equiv & \text{“ Property of hiding ”} \\ & Wait\ 1; (Stop \sqcap Wait\ 1) \end{array}$$

Using priority we can model asynchronous communication without any nondeterminism:

$$Send(ch,e) \mathrel{\widehat{=}}_{pSC} Wait\ 1 \overset{\leftarrow}{\Box} (ch!e \xrightarrow{1} Skip)$$

[1] Two steps in this proof have no support in the laws presented in this work. The first one “Synchronising on waiting”, is a conclusion from two other properties

$$Wait\ 1; A \ ||\ Wait\ 1; B = Wait\ 1; (A \ ||\ B)$$

, and

$$Wait\ 1; A \ ||\ b \rightarrow B = Wait\ 1; (A \ ||\ B \rightarrow B)$$

. The second step “Property of external choice” is based on a property of CSP $((Skip \Box a \rightarrow Skip) \ ||\ a \rightarrow Skip) \setminus \{a\} = Skip \sqcap Stop$. In CSP it can be easily proven using the UTP semantics from [HH98].

$$\begin{array}{ll} & ((Skip \Box a \rightarrow Skip) \ ||\ a \rightarrow Skip) \setminus \{a\} \\ =_{CSP} & (((Skip \wedge \neg Stop) \vee (a \rightarrow Skip \wedge \neg Stop)) \ ||\ a \rightarrow Skip) \setminus \{a\} \\ =_{CSP} & ((Skip \wedge \neg Stop) \ ||\ a \rightarrow Skip) \setminus \{a\} \\ & \vee ((a \rightarrow Skip \wedge \neg Stop) \ ||\ a \rightarrow Skip) \setminus \{a\} \\ =_{CSP} & (Skip \ ||\ a \rightarrow Skip) \setminus \{a\} \\ & \vee Skip \\ =_{CSP} & Stop \sqcap Skip \end{array}$$

The described action can only communicate on the first clock cycle. If there is no node willing to listen then the action will terminate after the first clock tick.

$$
\begin{array}{ll}
 & \Big(\ Send(ch,e) \mid\mid (Wait\ 1; ch?x \xrightarrow{1} Skip)\ \Big) \setminus \{ch\} \\
\equiv & \text{“ } Send \text{ definition with priority ”} \\
 & \Big(\ (Wait\ 1 \overset{\leftarrow}{\Box} (ch!e \xrightarrow{1} Skip)) \mid\mid (Wait\ 1; ch?x \xrightarrow{1} Skip)\ \Big) \setminus \{ch\} \\
\equiv & \text{“ Synchronising on waiting ”} \\
 & Wait\ 1; \Big(\ (Skip \overset{\leftarrow}{\Box} (ch!e \xrightarrow{1} Skip)) \mid\mid ch?x \xrightarrow{1} Skip\ \Big) \setminus \{ch\} \\
\equiv & \text{“ Skip as a high priority option ”} \\
 & Wait\ 1; \Big(\ Skip \mid\mid ch?x \xrightarrow{1} Skip\ \Big) \setminus \{ch\} \\
\equiv & \text{“ No synchronisation is possible ”} \\
 & Wait\ 1; Stop
\end{array}
$$

In case of a receiving node we need untimed communication to reason about collisions and again prioritised choice to make it nonblocking (as in the example above). Collisions appear when more than one communications occur at the same channel, at the same time.

$$
Receive(ch,x) \mathrel{\widehat{=}}_{pSC} \quad ch?x \rightarrow \left(\begin{array}{l} ch?x \xrightarrow{1} Collision \\ \overset{\rightarrow}{\Box} \\ Wait\ 1 \end{array} \right) \overset{\leftarrow}{\Box} Wait\ 1
$$

It is possible to describe collisions without untimed communication, where for each channel and every possible source of communication we create a sub-channel *channelName.sourceID* (one source can not produce colliding communication). We then collect the messages of all the sub-channels and count how many were received ($x1+x2+x3+...+xn$).

$$
\begin{array}{ll}
Receive_{timed}(ch,x) \mathrel{\widehat{=}} & x1,x2,x3,...,xn := 0; \\
 & \left(\left(\begin{array}{l} ch.1?x \xrightarrow{1} x1 := 1 \\ \mid\mid\mid ch.2?x \xrightarrow{1} x2 := 1 \\ \mid\mid\mid ch.3?x \xrightarrow{1} x3 := 1 \\ ... \\ \mid\mid\mid ch.n?x \xrightarrow{1} xn := 1 \end{array} \right) \overset{\leftarrow}{\Box} Wait\ 1 \right); \\
 & (Collision \lhd (x1+x2+x3+...+xn > 1) \rhd Skip)
\end{array}
$$

While $Receive_{timed}$ implements[2] *Receive* it is engaging many variables and for each is generating a sub-channel. Because communication is a basic building block of any specification, the use of $Receive_{timed}$ implies multiplication of size and complexity of the specification.

7.2.1 Safety

In the given definition of *Receive* we are using untimed communication which might introduce complications presented throughout this work. However the given definition is in the authors' opinion the best example of how the untimed prefix can be safely used with a great deal of advantages for both the readability of speci-

[2] Not in the strict, refinement meaning

fication and ease of reasoning. First of all if we analyse the *Recive* action we will notice that even with the untimed communication it is strong prefix healthy ([isSPH:def]:p. 47). This occurs because we sequentially compose two of the same input channels which can be shown to be equivalent to a process that uses only timed communication (and so is strong prefix healthy, see section4.4).

$$a?x \rightarrow a?x \xrightarrow{1} P = ((a?x \xrightarrow{1} Skip) \;|||\; (a?x \xrightarrow{1} Skip)); P$$

This property was used in the definition of $Recieve_{timed}$ and the only difference between the two definitions are the changes to the state that the second is introducing. This in fact is the other reason why we do prefer to use *Recieve*. The second definition, by introducing changes to state, has to be treated with caution not to cause side effects (overwriting global variables, changing behaviour of following processes by changes to the state). The size and complexity of it also means that more proof steps needs to be performed whenever it is used. Assuming that receiving will account for a significant part of the specification this will make substantial impact on the verification/reasoning effort.

Recieve is an example of how we can define a simple and safe macro using untimed communication. This is also the best example justifying an effort of reasoning about it.

7.3 Fixed-Priority Non Preemptive Scheduling

Fixed-priority non preemptive scheduling [YBB09] is a scheduling method commonly used in the real-time systems. With fixed priority non preemptive scheduling, the scheduler ensures that at any given time, the processor executes the highest priority task of all those tasks that are currently ready to execute. Tasks once started can not be interrupted. In the specification presented below tasks with the same priority are chosen nondeterministically, thanks to that an implementation can use any available algorithm.

In our example system we have a number of processes with unique ID's and fixed priorities. The given processes are running in parallel and have no possibility to interact with each other (are interleaved). Also in the system there is a single resource that the processes have to share, and a scheduler that decides which of the processes gets access to the resource based on its priority.

$$ExampleSystem =$$
$$\left(\begin{array}{l} (Process_{1,3} \;|||\; Process_{2,5} \;|||\; Process_{3.2}) \\ \| \\ \left(\begin{array}{l} Scheduler \\ \quad \| \\ \{|resourceReserved, resourceFree|\} \\ Resource \end{array} \right) \setminus \{|\, resourceReserved \,|\} \end{array} \right) \setminus \{|\, callResource, resourceFree \,|\}$$

Every process has to satisfy the following specification. A process with the given *priority* and *id* asks for access to the resource ($callResource!priority!id$) then works with the resource using the *resourceComm* communication channel. Finally when done signals that he no longer needs access to the resource $resourceFree.id$ and terminates.

$$Process_{id,priority} = \left(\begin{array}{l} callResource!priority!id \rightarrow (Work(id)) \setminus (All \setminus \{|\, resourceComm.id \,|\}); \\ resourceFree.id \rightarrow Skip \end{array} \right)$$

Resource has to satisfy the following specification. First it receives an information from the scheduler with an *id* of the process that it supposed to work with ($resourceReserved?id$). And then it cooperates with the

process until the *resourceFree.id* command is issued.

$$Resource = \left(\begin{array}{l} resourceReserved?id \rightarrow (WorkWith(id))\backslash(All\backslash\{|\ resourceComm.id\ |\}); \\ resourceFree.id \rightarrow Resource \end{array} \right)$$

Finally we can define the scheduler. The scheduler awaits requests for the resource from the processes. If only one request is made during the clock cycle then the scheduler will choose it. If more requests are made during the clock cycle then the scheduler will nondeterministically choose one from the available, top priority options. Once the scheduler chooses a process it informs the resource which one is it (*resourceReserved*!*id*) and awaits the signal that the resource is free again (*resourceFree.id*).

$$\begin{aligned} Scheduler &= \overset{\leftarrow}{\underset{i\in\langle 1..n\rangle}{\Box}} (callResource.i?id \rightarrow resourceReserved!id \rightarrow resourceFree.id \rightarrow Scheduler) \\ &= (callResource.1?id \rightarrow resourceReserved!id \rightarrow resourceFree.id \rightarrow Scheduler) \\ &\quad \overset{\leftarrow}{\Box} (callResource.2?id \rightarrow resourceReserved!id \rightarrow resourceFree.id \rightarrow Scheduler) \\ &\quad \overset{\leftarrow}{\Box} (callResource.3?id \rightarrow resourceReserved!id \rightarrow resourceFree.id \rightarrow Scheduler) \\ &\quad \ldots \\ &\quad \overset{\leftarrow}{\Box} (callResource.n?id \rightarrow resourceReserved!id \rightarrow resourceFree.id \rightarrow Scheduler) \end{aligned}$$

The given specification of the scheduler is focused on two aspects: safety and timing. Safety means that the scheduler works as intended while timing stands for minimal delays that it introduces and in that aspect the proposed solution is rather extreme.

Resolving priority and reserving the resource is performed within a clock cycle. Similar freeing of the resource introduces no delays. In other words a process can start its work with the resource at the same clock cycle as the previous process has finished. In an extreme case when the communication between a processes and the resource is only a single question-answer (taking one clock cycle each) it is possible to handle a new process in every two clock cycles.

The only timing constrain that we might face while implementing the given specification is the increasing propagation time of the hardware logic, depending on the number of possible priorities. In the worst case a clock tick might be necessary after the *callResource* event to reduce the need to slow down the hardware clock.

7.3.1 Verification

We want to show that an example system is safe by showing that it is equivalent to a system using timed communication only. To reduce the size of the proof we consider a system with only two processes, with different priorities.

First we reduce the specification of the scheduler interacting with the resource (*SandR*):

$$
\begin{array}{ll}
 & SandR \\
\equiv & \\
 & \left(Scheduler \underset{\{|resourceReserved, resourceFree|\}}{||} Resource \right) \setminus \{| resourceReserved |\} \\
\equiv & \text{“ def of Scheduler and Resource ”} \\
 & \left(\begin{array}{l} \left(\begin{array}{l} \overset{\leftarrow}{\Box}_{i \in \langle 1..n \rangle} (callResource.i?id \rightarrow resourceReserved!id \\ \rightarrow resourceFree.id \rightarrow Scheduler) \end{array} \right) \\ \underset{\{|resourceReserved, resourceFree|\}}{||} \\ \left(\begin{array}{l} resourceReserved?id \\ \rightarrow (WorkWith(id)) \setminus (All \setminus \{| resourceComm.id |\}); \\ resourceFree.id \rightarrow Resource \end{array} \right) \end{array} \right) \setminus \{| resourceReserved |\} \\
\equiv & \text{“ [StepLaw1]:p. 73, [Hide:law2]:p. 72 ”} \\
 & \overset{\leftarrow}{\Box}_{i \in \langle 1..n \rangle} (callResource.i?id \rightarrow \\
 & \left(\begin{array}{l} \left(\begin{array}{l} (resourceReserved!id \\ \rightarrow resourceFree.id \rightarrow Scheduler) \end{array} \right) \\ \underset{\{|resourceReserved, resourceFree|\}}{||} \\ \left(\begin{array}{l} resourceReserved?id \\ \rightarrow (WorkWith(id)) \setminus (All \setminus \{| resourceComm.id |\}); \\ resourceFree.id \rightarrow Resource \end{array} \right) \end{array} \right) \setminus \{| resourceReserved |\} \\
 &) \\
\equiv & \text{“ [StepLaw2]:p. 74 ”} \\
 & \overset{\leftarrow}{\Box}_{i \in \langle 1..n \rangle} (callResource.i?id \rightarrow \\
 & \left(\begin{array}{l} resourceReserved.id \rightarrow \\ \left(\begin{array}{l} resourceFree.id \rightarrow Scheduler \\ \underset{\{|resourceReserved, resourceFree|\}}{||} \\ \left(\begin{array}{l} WorkWith(id) \setminus (All \setminus \{| resourceComm.id |\}); \\ resourceFree.id \rightarrow Resource \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{| resourceReserved |\} \\
 &) \\
\equiv & \text{“ [Hide:law1]:p. 72 ”} \\
 & \overset{\leftarrow}{\Box}_{i \in \langle 1..n \rangle} (callResource.i?id \rightarrow \\
 & \left(\begin{array}{l} resourceFree.id \rightarrow Scheduler \\ \underset{\{|resourceReserved, resourceFree|\}}{||} \\ \left(\begin{array}{l} WorkWith(id) \setminus (All \setminus \{| resourceComm.id |\}); \\ resourceFree.id \rightarrow Resource \end{array} \right) \end{array} \right) \setminus \{| resourceReserved |\} \\
 &) \\
\equiv & \text{“ [StepLaw1]:p. 73 ”}
\end{array}
$$

$$
\begin{aligned}
\equiv \quad & \text{“ [StepLaw1]:p. 73 ”} \\
& \overset{\leftarrow}{\underset{i\in\langle 1..n\rangle}{\Box}} (callResource.i?id \rightarrow \\
& \left(\begin{array}{l} WorkWith(id) \setminus (All \setminus \{\!|\ resourceComm.id\ |\!\}); \\ \left(\begin{array}{c} resourceFree.id \rightarrow Scheduler \\ \underset{\{\!|resourceReserved, resourceFree|\!\}}{||} \\ \left(\ resourceFree.id \rightarrow Resource \ \right) \end{array} \right) \end{array} \right) \setminus \{\!|\ resourceReserved\ |\!\} \\
&) \\
\equiv \quad & \text{“ [Hide:law2]:p. 72 ”} \\
& \overset{\leftarrow}{\underset{i\in\langle 1..n\rangle}{\Box}} (callResource.i?id \rightarrow WorkWith(id) \setminus (All \setminus \{\!|\ resourceComm.id\ |\!\}); \\
& \left(\begin{array}{c} resourceFree.id \rightarrow Scheduler \\ \underset{\{\!|resourceReserved, resourceFree|\!\}}{||} \\ resourceFree.id \rightarrow Resource \end{array} \right) \setminus \{\!|\ resourceReserved\ |\!\} \\
&) \\
\equiv \quad & \text{“ [StepLaw2b]:p. 74, [Hide:law2]:p. 72 ”} \\
& \overset{\leftarrow}{\underset{i\in\langle 1..n\rangle}{\Box}} (callResource.i?id \rightarrow WorkWith(id) \setminus (All \setminus \{\!|\ resourceComm.id\ |\!\}); \\
& resourceFree.id \rightarrow \\
& \left(Scheduler \underset{\{\!|resourceReserved, resourceFree|\!\}}{||} Resource \right) \setminus \{\!|\ resourceReserved\ |\!\} \\
&) \\
\equiv \quad & \\
& \overset{\leftarrow}{\underset{i\in\langle 1..n\rangle}{\Box}} \left(\begin{array}{l} callResource.i?id \rightarrow WorkWith(id) \setminus (All \setminus \{\!|\ resourceComm.id\ |\!\}); \\ resourceFree.id \rightarrow SandR \end{array} \right)
\end{aligned}
$$

After changing the scope of our specification and treating the resources and the scheduler as one system we no longer see the internal communication between them.

We can now analyse the safety of the system along with the two example processes - $Process_{1,1}$ and $Process_{2,2}$.

$$
\begin{array}{ll}
 & ExampleSystem \\
\equiv & \text{“ definition ”} \\
 & \left(\begin{array}{l}
(Process_{1,1} \;|||\; Process_{2,2}) \\
\;||\; \\
\left(\begin{array}{l} Scheduler \\ \quad \underset{\{|resourceReserved, resourceFree|\}}{||} \\ Resource \end{array}\right) \setminus \{|\, resourceReserved \,|\}
\end{array}\right) \\
 & \setminus \{|\, callResource, resourceFree \,|\} \\
\equiv & \text{“ RandS def ”} \\
 & \left(\begin{array}{l}
(Process_{1,1} \;|||\; Process_{2,2}) \\
\;||\; \\
\underset{i \in \langle 1..n \rangle}{\overset{\leftarrow}{\Box}} \left(\begin{array}{l} callResource.i?id \rightarrow WorkWith(id) \setminus (All \setminus \{|\, resourceComm.id \,|\}); \\ resourceFree.id \rightarrow SandR \end{array}\right)
\end{array}\right) \\
 & \setminus \{|\, callResource, resourceFree \,|\} \\
\equiv & \text{“ Process definition ”} \\
 & \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} callResource!1!1 \rightarrow (Work(1)) \setminus (All \setminus \{|\, resourceComm.1 \,|\}); \\ resourceFree.1 \rightarrow Skip \end{array}\right) \\
|||\; \left(\begin{array}{l} callResource!2!2 \rightarrow (Work(2)) \setminus (All \setminus \{|\, resourceComm.2 \,|\}); \\ resourceFree.2 \rightarrow Skip \end{array}\right)
\end{array}\right) \\
\;||\; \\
\underset{i \in \langle 1..n \rangle}{\overset{\leftarrow}{\Box}} \left(\begin{array}{l} callResource.i?id \rightarrow WorkWith(id) \setminus (All \setminus \{|\, resourceComm.id \,|\}); \\ resourceFree.id \rightarrow SandR \end{array}\right)
\end{array}\right) \\
 & \setminus \{|\, callResource, resourceFree \,|\} \\
\equiv & \text{“ [StepLaw3]:p. 74 ”} \\
 & \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} Work(1) \setminus (All \setminus \{|\, resourceComm.1 \,|\}); \\ resourceFree.1 \rightarrow Skip \end{array}\right) \\
|||\; \left(\begin{array}{l} callResource!2!2 \rightarrow (Work(2)) \setminus (All \setminus \{|\, resourceComm.2 \,|\}); \\ resourceFree.2 \rightarrow Skip \end{array}\right)
\end{array}\right) \\
\;||\; \\
\left(\begin{array}{l} WorkWith(1) \setminus (All \setminus \{|\, resourceComm.1 \,|\}); resourceFree.1 \rightarrow SandR \end{array}\right)
\end{array}\right) \\
 & \setminus \{|\, callResource, resourceFree \,|\} \\
\equiv & \text{“ [StepLaw4]:p. 75 applied iteratively, state changes of } Work \;||\; WorkWith \text{ are irrelevant ”} \\
 & \left(\begin{array}{l}
\left(\begin{array}{l} Work(1) \setminus (All \setminus \{|\, resourceComm.1 \,|\}) \;|| \\ WorkWith(1) \setminus (All \setminus \{|\, resourceComm.1 \,|\}) \end{array}\right); \\
\left(\begin{array}{l}
\left(\begin{array}{l} resourceFree.1 \rightarrow Skip \end{array}\right) \\
|||\; \left(\begin{array}{l} callResource!2!2 \rightarrow (Work(2)) \setminus (All \setminus \{|\, resourceComm.2 \,|\}); \\ resourceFree.2 \rightarrow Skip \end{array}\right)
\end{array}\right) \\
\;||\; \\
\left(\begin{array}{l} resourceFree.1 \rightarrow SandR \end{array}\right)
\end{array}\right) \\
 & \setminus \{|\, callResource, resourceFree \,|\} \\
\equiv & \text{“ [Hide:law2]:p. 72 ”}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \text{“ [Hide:law2]:p. 72 ”} \\
& \left(\begin{array}{l} Work(1) \setminus (All \setminus \{\!| \, resourceComm.1 \, |\!\}) \\ || \\ WorkWith(1) \setminus (All \setminus \{\!| \, resourceComm.1 \, |\!\}) \end{array} \right); \\
& \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} resourceFree.1 \rightarrow Skip \end{array} \right) \\ ||| \left(\begin{array}{l} callResource!2!2 \rightarrow (Work(2)) \setminus (All \setminus \{\!| \, resourceComm.2 \, |\!\}); \\ resourceFree.2 \rightarrow Skip \end{array} \right) \end{array} \right) \\ || \\ \left(\begin{array}{l} resourceFree.1 \rightarrow SandR \end{array} \right) \end{array} \right) \\
& \setminus \{\!| \, callResource, resourceFree \, |\!\} \\
\equiv & \text{“ [StepLaw4]:p. 75, [Hide:law1]:p. 72 , [Interleaving: unit law]:p. 61 ”} \\
& \left(\begin{array}{l} Work(1) \setminus (All \setminus \{\!| \, resourceComm.1 \, |\!\}) \\ || \\ WorkWith(1) \setminus (All \setminus \{\!| \, resourceComm.1 \, |\!\}) \end{array} \right); \\
& \left(\begin{array}{l} \left(\begin{array}{l} callResource!2!2 \rightarrow (Work(2)) \setminus (All \setminus \{\!| \, resourceComm.2 \, |\!\}); \\ resourceFree.2 \rightarrow Skip \end{array} \right) \\ || \\ SandR \end{array} \right) \\
& \setminus \{\!| \, callResource, resourceFree \, |\!\} \\
\equiv & \text{“ Analogous to the proof so far ”} \\
& \left(\begin{array}{l} \left(\begin{array}{l} (Work(1)) \setminus (All \setminus \{\!| \, resourceComm.1 \, |\!\}) \end{array} \right) \\ || \\ \left(\begin{array}{l} WorkWith(1) \setminus (All \setminus \{ \, resourceComm.1 \, \}); \end{array} \right) \end{array} \right); \\
& \left(\begin{array}{l} \left(\begin{array}{l} (Work(2)) \setminus (All \setminus \{\!| \, resourceComm.2 \, |\!\}) \end{array} \right) \\ || \\ \left(\begin{array}{l} WorkWith(2) \setminus (All \setminus \{ \, resourceComm.2 \, \}); \end{array} \right) \end{array} \right); \\
& (Skip \, || \, SandR) \setminus \{\!| \, callResource, resourceFree \, |\!\} \\
\equiv & \text{“ [Parallel: skip and comm]:p. 62 ”} \\
& \left(\begin{array}{l} \left(\begin{array}{l} (Work(1)) \setminus (All \setminus \{\!| \, resourceComm.1 \, |\!\}) \end{array} \right) \\ || \\ \left(\begin{array}{l} WorkWith(1) \setminus (All \setminus \{ \, resourceComm.1 \, \}); \end{array} \right) \end{array} \right); \\
& \left(\begin{array}{l} \left(\begin{array}{l} (Work(2)) \setminus (All \setminus \{\!| \, resourceComm.2 \, |\!\}) \end{array} \right) \\ || \\ \left(\begin{array}{l} WorkWith(2) \setminus (All \setminus \{ \, resourceComm.2 \, \}); \end{array} \right) \end{array} \right); Stop
\end{array}
$$

As we can see we were able to reduce *ExampleSystem* to a process that sequentially executes two processes with respect to their priorities. No delays are introduced between the execution of the two processes and no untimed communication is being used. That way we reduced the safety and usability of the system to correctness of cooperation between the given processes and the resource.

7.4 Conclusions

In this chapter three case studies making extensive use of priority have been presented. As expected, priority proved to be useful in describing the notion of preference in the logical structure of the processes (prialt construct in Handel-C, priority in the scheduling method). More interestingly, priority proves ideal for describing broadcasting/assynchronous type of communication, which is unnatural for a CSP based language (signal variable in Handel-C, broadcasting with possible collisions in WSN-s).

Another interesting observation is how important untimed communication proved in all of the case studies. In Handel-C it is the only way of describing the timing details of its operators, and while it makes the specification "unsafe" it exactly mirrors the possibilities of writing unimplementable programs in Handel-C. In the case of WSN-s, untimed communication allows to greatly reduce the size and complexity of the specification (probably reducing the complexity of verification by a polynomial factor). It is worth noting that in the case of WSN-s, untimed communication is hidden behind predefined macros that make it safe to use (no causality problems are possible). From that we can conclude the possibility of using untimed communication as a tool (building block) for defining operators and making it invisible for the end user of the language. Finally the scheduling method also uses the untimed communication to achieve time efficiency that otherwise would be impossible. The description of the scheduler is then proved to be safe to use (no causality problems are possible), by showing that it implements a system where no untimed communication is used.

Chapter 8

Conclusions and Future Work

This research originally aimed to finalise the UTP denotational semantics of slotted-*Circus* by adapting it, so that it would be able to model the behaviour of a large subset of Handel-C, verify it and explore links between slotted-*Circus* and *Circus*. Once slotted-*Circus* had been defined the search for a connection between it and the untimed theory lead us to the discovery of priority. Since then the research branched into many directions, producing many surprising conclusions and rising even more interesting questions. It was the author's decision not to choose any of the safe and easy solutions (timed prefix, section 4.2) but rather to think about priority with no restrictions at all, discuss the consequences and possible solutions, raise questions and show reasons for pursuing them. It still needs to be determined which questions are worth answering, and which branches of this research are worth following. The conclusions drawn below will hopefully help other researchers making that decision.

8.1 The UTP Semantics of slotted-*Circus*

slotted-*Circus* is a language with very complex UTP semantics, which in the last few years was used as an argument against UTP . The truth is though that slotted-*Circus* has a very complicated behaviour that needs to be described. State, stability and precise timing issues have to be addressed in every definition and any denotational semantics of it will be complex and difficult to process. slotted-*Circus* has reached a point where denotational semantics can no longer be used as a reference manual, in the same way that details of compilation are not used to teach C.

8.1.1 Definition of Priority

During this research the denotational semantics of prioritised choice were defined based on a very simple verbal definition:

> *When lower and higher priority options are willing to perform an event (or terminate) at the same time then the higher priority option should be chosen. Else the prioritised choice behaves like an external choice.*

, and on the observation that the whole theory of how parallel priorities should be synchronised was already defined for the refusals sets.[1]

The simplicity of this observation led to a definition of priority that was formulated even before any of the desired laws were listed and was left unchanged since then. Often during this research, based on the author's understanding of priority, the desired laws that were being formulated, were contradictory to the results given

[1] Overloading refusals sets with the task of solving and enforcing priority restrictions did not imply any changes to how the refusals mechanism was defined (for details see subsection 5.3.2).

by the definition. Careful examination of the proof steps always proved the superiority of the definition above the author's "common sense".

A good example of an "obvious" law of prioritised choice is the interaction between the external and prioritised choice.

$$(a \rightarrow Skip \overleftarrow{\Box} b \rightarrow Skip) \,||\, (a \rightarrow Skip \,\Box\, b \rightarrow Skip) = a \rightarrow Skip \overleftarrow{\Box} b \rightarrow Skip$$

While intuitively the preference should propagate through the parallel construct and be enforced on external choice, this law was contradicted by the denotational semantics of the parallel construct[2]. Careful examination of the counter example led to an observation that this law was in fact unsound with respect to some other desired laws of priority.

$$\begin{aligned} & (a \rightarrow Skip \overleftarrow{\Box} b \rightarrow Skip) \,||\, (a \rightarrow Skip \,\Box\, b \rightarrow Skip) \\ = \;& (a \rightarrow Skip \overleftarrow{\Box} b \rightarrow Skip) \,||\, ((a \rightarrow Skip \overleftarrow{\Box} b \rightarrow Skip) \,||\, (a \rightarrow Skip \overrightarrow{\Box} b \rightarrow Skip)) \\ = \;& ((a \rightarrow Skip \overleftarrow{\Box} b \rightarrow Skip) \,||\, (a \rightarrow Skip \overleftarrow{\Box} b \rightarrow Skip)) \,||\, (a \rightarrow Skip \overrightarrow{\Box} b \rightarrow Skip) \end{aligned}$$

$A \,||\, A = A$ for deterministic A[3].

$$\begin{aligned} = \;& (a \rightarrow Skip \overleftarrow{\Box} b \rightarrow Skip) \,||\, (a \rightarrow Skip \overrightarrow{\Box} b \rightarrow Skip) \\ = \;& a \rightarrow Skip \,\Box\, b \rightarrow Skip \end{aligned}$$

This and many other examples prove the necessity of an effort to formally define the behaviour of a language and how important it is to verify even the simplest of laws.

Also it is worth mentioning that the denotational semantics of priority proved invaluable in understanding of how preference lists can be synchronised[4], even if the nontransitivity of preference is proving problematic for many years now [Con85], [BHM88].

8.2 Priority

8.2.1 Connection between Time and Priority

While describing the `prialt` construct of Handel-C, which was one of the original targets of this work, the need for the prioritised choice was identified when trying to define a link between slotted-*Circus* and *Circus*. Based on works like [Ros98] or [She06] we could conclude that modelling a timed behaviour using an untimed theory is possible simply by modelling a *tick* (or *tock* as presented in [Ros98]) event. Unfortunately it is not enough. While the *tick* event is good for synchronisation of parallel processes with the clock, in the standard untimed theory there are no tools (operators) to enforce the maximal urgency principle. To do so *tick* should be treated asymmetrically to any other event, and whenever a choice is given between them the event should be chosen. Unfortunately, CSP does not have asymmetric operators other than guards. In other words, in order to properly implement the behaviour of a clock the preference needs to be modelled using state. Using guards would require checking what events are available, analysing them and finally making a decision. Another argument which explains why the ability to model time with an untimed theory is very limited is the presence

[2] To be exact, the presented theorem was contradicted by the way that refusals sets of parallel processes are synchronised. While many aspects of parallel construct varies, the synchronisation of refusals sets is the same for all of the CSP-based languages.

[3] *"All laws for nondeterministic processes are true for deterministic processes as well. But deterministic processes obey some additional laws, for example"*

$$P \,||\, P = P$$

[Hoa85a]p.74

[4] All the laws and intuitive explanations presented in section 6.1 have been based on the denotational semantics of prioritised slotted-*Circus* and multiple sketch proofs

of only one refusals set per trace in the semantics of the untimed theory. In order to enforce the maximal urgency rule a separate refusals set is necessary for every clock cycle (in case of slotted-*Circus* every *slot* has a separate refusals set).

Based on the above, the initial attempt to define priority was by defining it for theories such as *Circus* or CSP. Unfortunately a second discovery was made where priority, as defined in this thesis, could only be defined for a timed theory because it needs a precise deadline for when it stops waiting for the availability of events and makes a choice (see sec 5.2 and 5.4).

A final observation arises from the work on the semantics of `prialt` in Handel-C, where the idea of micro-slots is used to model its behaviour [But11a]. The conclusion is that priority could be modelled by introducing micro delays to events available within the same clock cycle based on its priority, and then choosing an event on the first-come, first-served basis.

To conclude, priority (as presented in this thesis) allows us to model time by being able to describe the asymmetric behaviours of *tick* in respect to other events. On the other hand, priority can only be defined when time is present so the deadline for the choice can be set, and priority can easily be described with a theory that has two levels of time granularity.

8.2.2 More Priority - Less State

Because CSP is Turing-complete it can not claimed that with priority it would be possible to describe a behaviour that is not be possible to be described using CSP, *Circus* or slotted-*Circus*. However, it can be claimed that by providing an asymmetric operator, situations that else would be very complex to reason in theories without priority, can easily be described. Every case study presented in this thesis could be addressed with other theories, but they would imply an excessive use of variables and guards to be modelled. By being able to model complicated examples using only algebraic properties of priority slotted-*Circus* is closer to the idea of process algebras which reduces the impact that state has on verification.

8.2.3 Untimed communication

Immediate causality is a complex issue that can be addressed in three possible ways. The simplest solution is to enforce timed communication. Even with that restriction prioritised slotted-*Circus* is still a very expressive language. It allows to describe asynchronous communication (accepting the increased size of the specification) (Sec. 7.2), and it allows to model scheduling algorithms(Sec. 7.3) as long as the reduced time performance of the described solution is acceptable.

The second solution is to use the untimed communication only to define new operators, like asynchronous communication in WSN-s 7.2, or small, time critical parts of specification like in the scheduler example (Sec. 7.3). In that case careful analysis has to be performed to make sure that none of the undesired causality loops or cross-communication will be possible. It is unclear how difficult (if possible) it will be to implement the safety analysis in the tool support.

Finally, it is possible to use untimed communication which would allow prioritised slotted-*Circus* to describe even a very low level hardware design. Unfortunately using untimed communication would imply the need for static checking similar to the one performed by the Handel-C compiler.

8.2.4 Applications

Some of the most interesting results presented in this thesis are examples where priority can easily describe complex behaviours. The most obvious use cases of priority are priority scheduling, interrupt mechanisms and in Real Time Operating Systems [DGM09]. There is no doubt that priority can be directly used there. What is interesting though is the low level hardware that is able to easily implement priority and that priority is a very desirable operator for its ability to save time in the often used time critical routines. Also surprising is the ease with which we can define asynchronous communication/broadcasting, or collisions in WSN-a.

8.3 Future Work

8.3.1 Time granularity

It is possible to extend slotted-*Circus* so it talks about more levels of granularity of time (micro and macro clock-ticks). I can be achieved in two ways. First by providing a new denotational semantics with more trace layers (for example adding "slotss" layer equivalent to slots of slots). Unfortunately this would complicate the already complex semantics and would require a substantial effort in reverifying it. The second way would be to use the property mentioned earlier that by using priority it is possible define models of time. Using prioritised slotted-*Circus* we could implement a *tick* event with desired properties.

The application of this solution could be for example the ability to reason about signal propagation times in synchronous hardware designs. This in return would give us the ability to reason how large can a part of hardware logic be in order not to slow down the whole synchronous design, and yet perform as much computations as possible within a single clock cycle. With granular time, there are no longer any reasons to use untimed communication which would make the language easier to use.

8.3.2 New target hardware language

Depending on the decision of Mentor Graphics about a future of Handel-C it might be necessary to find a new target implementation language for slotted-*Circus*. One of the possibilities could be Esterel [BCG86] which is in many aspects similar to Handel-C. It is also possible to choose some lower level hardware description language (like VHDL) but a proper feasibility study needs to be performed here.

8.3.3 Proof methods

As mentioned before most of the laws for prioritised slotted-*Circus* have been developed based on the given denotational semantics. Unfortunately ways in which the author was able to do it could not be formalised and so can not be completely relied on. At the other hand the formal proofs presented in the appendix along with proofs in the slotted-*Circus* tech report [BG09] are of substantial length and demand considerable investment of time and effort for even the most basic properties. A good example of a law that is not feasible to be proven with the current methods is one of the most important laws of priority.

$$
\begin{aligned}
P \,||\, Q \quad := \quad & \Box_{(i,j)\in TP}(p_i \rightarrow (P_i \parallel Q_j)) \overleftarrow{\Box} \\
& ((p_1 \rightarrow P_1 \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_{i-1} \rightarrow P_{i-1} \overleftarrow{\Box} p_{i+1} \rightarrow P_{i+1} \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_n \rightarrow P_n) \\
& \parallel (q_1 \rightarrow Q_1 \overleftarrow{\Box} \ldots \overleftarrow{\Box} q_{j-1} \rightarrow Q_{j-1} \overleftarrow{\Box} q_{j+1} \rightarrow Q_{j+1} \overleftarrow{\Box} \ldots \overleftarrow{\Box} q_n \rightarrow Q_n))
\end{aligned}
$$

Where,

$$TP \mathrel{\widehat{=}} \{(i,j) \bullet p_i = q_j \wedge \forall k,l \bullet (p_k \neq q_l \vee p_k < p_i \vee q_l < q_j)\}$$

, and

$$P = p_1 \rightarrow P_1 \overleftarrow{\Box} p_2 \rightarrow P_2 \overleftarrow{\Box} \ldots \overleftarrow{\Box} p_n \rightarrow P_n$$

$$Q = q_1 \rightarrow Q_1 \overleftarrow{\Box} q_2 \rightarrow Q_2 \overleftarrow{\Box} \ldots \overleftarrow{\Box} q_n \rightarrow Q_n$$

, where P and Q are strong prefix healthy.

For that reason the proof methods available for slotted-*Circus* needs to be improved. This could achieved by the definition of numerous intermediate laws and increased number of basic actions.

8.3.4 Tool support

During this research it has become clear that proving properties, handling the library of available laws and proofs and most of all the reusability of performed proofs is not feasible. While working on the denotational semantics of slotted-*Circus* some definitions were redefined multiple times, notations were being changed to make the semantics more organised and new, more useful low level operators were being introduced in order to improve the proof methods. Every time this happened many available proofs were made obsolete and in many cases the simple "replace text" command was not enough to update them. Also while the proof steps are usually very simple the bulky definitions implies that often only few simple steps fit on one page which lead to simple "typo" mistakes being undetected. This situation led to the development of a proof assistant/theorem prover Saoithín [But10]that would maintain the library of proofs, detect when a change in the semantics causes a proof to be obsolete, try to redo the proof steps with new definitions and finally would take care of maintaining the large amount of text that is being generated by the proof. In the future the results presented in this thesis and the tech report will need to be exported and verified using Saoithín.

While Saoithín can already advise on possible proof steps it is possible that it will provide a connection to an external, fully automatic theorem prover or proof checker. An interesting way could be to try to translate it to the TPTP syntax [SS97] which is becoming a standard syntax for many of the modern automatic provers. The translation should be possible because the UTP semantics is based on the second order logic. If the translation from Saoithín to TPTP could be achieved, then it should also be possible to export (once proven) the slotted-*Circus* process algebra and use the range of theorem provers as a tool support for the language. This would lead to a situation when the most powerful/supported/fitting theorem prover could be used on slotted-*Circus* with the minimum support effort from the slotted-*Circus* community.

Appendix A

Low level slotted-*Circus*

In here we present a list of various laws and properties of slotted-*Circus*. The list was extracted from the technical report [BG09], was not created during this research and is kept here only for referencing purposes.

A.1 History models

A.1.1 MSA

[MSA:HIST] $MSA\ E \mathrel{\widehat=} E \to \mathbb{N}_1$

[MSA:ACC:sig] $acc_{MSA} : E^* f \mathbb{P} E$

[MSA:ACC:def] $acc(bag) \mathrel{\widehat=} dom(bag)$

[MSA:ET:sig] $EQVTRC_{MSA} : E^* \leftrightarrow MSA\ E$

[MSA:ET:def] $EQVTRC(tr, bag) \mathrel{\widehat=} items(tr) = bag$

[MSA:ET:elems] $EQVTRC(tr, bag) \Rightarrow elems(tr) = acc(bag)$

[MSA:HN:null] $acc(hnull) = \{\ \}$

[MSA:pfx:sig] $\preceq_{MSA} : E^* \leftrightarrow E^*$

[MSA:pfx:def] $bag_1 \preceq bag_2 \mathrel{\widehat=} bag_1 \sqsubseteq bag_2$

[MSA:pfx:refl] $bag \preceq bag = \text{TRUE}$

[MSA:pfx:trans] $bag_1 \preceq bag_2 \land bag_2 \preceq bag_3 \Rightarrow bag_1 \preceq bag_3$

[MSA:pfx:anti-sym] $bag_1 \preceq bag_2 \land bag_2 \preceq bag_1 \Rightarrow bag_1 = bag_2$

[MSA:SN:pfx] $hnull \preceq bag$

[MSA:ET:pfx] $bag_1 \preceq bag_2$
$\equiv$
$\exists tr_1, tr_2 \bullet EQVTRC(tr_1, bag_1) \land EQVTRC(tr_2, bag_2) \land tr_1 \leq tr_2$

[MSA:sadd:sig] $sadd_{MSA} : E^* \times E^* f E^*$

[MSA:sadd:def] $sadd(bag_1, bag_2) \mathrel{\widehat{=}} bag_1 \oplus bag_2$

[MSA:sadd:events] $acc(sadd(bag_1, bag_2) = acc(bag_1) \cup acc(bag_2)$

[MSA:sadd:unit] $sadd(bag_1, bag_2) = bag_1 \equiv (bag_2 = hnull)$

[MSA:sadd:assoc] $sadd(bag_1, sadd(bag_2, bag_3)) = sadd(sadd(bag_1, bag_2), bag_3)$

[MSA:sadd:prefix] $bag \preceq sadd(bag, bag)$

[MSA:ssub:sig] $ssub_{MSA} : E^* \times E^* \to E^*$

[MSA:ssub:def] $ssub(bag_1, bag_2) \mathrel{\widehat{=}} bag_1 \ominus bag_2$

[MSA:ssub:pre] $pre\text{-}ssub(bag_1, bag_2) = bag_2 \preceq bag_1$

[MSA:ssub:events] $bag_2 \preceq bag_1 \land s' = ssub(bag_1, bag_2)$
$\Rightarrow acc(bag_1) \setminus acc(bag_2) \subseteq acc(s') \subseteq acc(bag_1)$

[MSA:ssub:self] $SSub(bag, bag) = hnull$

[MSA:ssub:nil] $SSub(bag, hnull) = bag$

[MSA:SSub:same] $bag \preceq bag'_a \land bag \preceq bag'_b \Rightarrow$
$ssub(bag'_a, bag, ref) = ssub(bag'_b, bag)$
$\equiv bag'_a = bag'_b$

[MSA:ssub:subsub] $bag_c \preceq bag_a \land bag_c \preceq bag_b$
$\land bag_b \preceq bag_a$
$\Rightarrow ssub(ssub(bag_a, bag_c), ssub(bag_b, bag_c))$
$= ssub(bag_a, bag_b)$

[MSA:sadd:ssub] $bag \preceq bag' \Rightarrow$
$sadd(bag, ssub(bag', bag)) = bag'$

[MSA:ssub:sadd] $ssub(sadd(bag_1, bag_2), bag_1) = bag_2$

[MSA:SNC:sig] $ssync_{MSA} : \mathbb{P} E f E^* \times E^* f \mathbb{P}(E^*)$

[MSA:SNC:one] $\forall s' \in ssync(cs)(s_1, hnull) \bullet acc(s') \subseteq acc(s_1) \setminus cs$

[MSA:SNC:only] $s' \in acc(ssync(cs)(s_1, s_2)) \Rightarrow acc(s') \subseteq acc(s_1) \cup acc(s_2)$

[MSA:SNC:sync] $s' \in acc(ssync(cs)(s_1, s_2)) \Rightarrow cs \cap acc(s') \subseteq cs \cap (acc(s_1) \cap acc(s_2))$

[MSA:SNC:assoc] $syncset(cs)(s_1)(ssync(cs)(s_2, s_3)) = syncset(cs)(s_3)(ssync(cs)(s_1, s_2))$

[MSA:SHid:sig] $shide_{MSA} : \mathbb{P}E \mathit{f} E^* \mathit{f} E^*$

[MSA:SHid:def] $shide(hdn)bag \mathrel{\widehat=} hdn \lhd bag$

[MSA:SHid:evts] $acc(shide(hdn)bag) = acc(bag) \setminus hdn$

A.1.2 *CTA*

A.2 Slotted-*Circus*—CTA Incarnation

This section presents the CTA theory of [She06, pp27–88] in the slotted-*Circus* framework:

[CTA:HIST] $CTA\ E \mathrel{\widehat=} E^*$

[CTA:ACC:sig] $acc_{CTA} : E^* \mathit{f} \mathbb{P}E$

[CTA:ACC:def] $acc(str) \mathrel{\widehat=} elems(str)$

[CTA:ET:sig] $EQVTRC_{CTA} : E^* \leftrightarrow CTA\ E$

[CTA:ET:def] $EQVTRC(tr, str) \mathrel{\widehat=} tr = str$

[CTA:ET:elems] $EQVTRC(tr, str) \Rightarrow elems(tr) = acc(str)$

[CTA:HN:sig] $hnull_{CTA} : E^*$

[CTA:HN:def] $hnull \mathrel{\widehat=} \langle\rangle$

[CTA:HN:null] $acc(hnull) = \{\}$

[CTA:pfx:sig] $\preceq_{CTA} : E^* \leftrightarrow E^*$

[CTA:pfx:def] $str_1 \preceq str_2 \mathrel{\widehat=} str_1 \leq str_2$

[CTA:pfx:refl] $str \preceq str = \text{TRUE}$

[CTA:pfx:trans] $str_1 \preceq str_2 \land str_2 \preceq str_3 \Rightarrow str_1 \preceq str_3$

[CTA:pfx:anti-sym] $str_1 \preceq str_2 \land str_2 \preceq str_1 \Rightarrow str_1 = str_2$

[CTA:SN:pfx] $hnull \preceq str$

[CTA:ET:pfx] $str_1 \preceq str_2$

$$\Rightarrow$$

$$\exists tr_1, tr_2 \bullet EQVTRC(tr_1, str_1) \land EQVTRC(tr_2, str_2) \land tr_1 \leq tr_2$$

[CTA:sadd:sig] $sadd_{CTA} : E^* \times E^* \mathit{f} E^*$

[CTA:sadd:def] $sadd(str_1, str_2) \mathrel{\widehat=} str_1 \frown str_2$

[CTA:sadd:events] $acc(sadd(str_1, str_2) = acc(str_1) \cup acc(str_2)$

[CTA:sadd:unit] $sadd(str_1, str_2) = str_1 \equiv (str_2 = hnull)$

[CTA:sadd:assoc] $sadd(str_1, sadd(str_2, str_3)) = sadd(sadd(str_1, str_2), str_3)$

[CTA:sadd:prefix] $str \preceq sadd(str, str)$

[CTA:ssub:sig] $ssub_{CTA} : E^* \times E^* \to E^*$

[CTA:ssub:def] $ssub(str_1, str_2) \mathrel{\widehat=} str_1 - str_2$

[CTA:ssub:pre] $pre\text{-}ssub(str_1, str_2) = str_2 \preceq str_1$

[CTA:ssub:events] $str_2 \preceq str_1 \land s' = ssub(str_1, str_2)$
$\Rightarrow acc(str_1) \setminus acc(str_2) \subseteq acc(s') \subseteq acc(str_1)$

[CTA:ssub:self] $SSub(str, str) = hnull$

[CTA:ssub:nil] $SSub(str, hnull) = str$

[CTA:SSub:same] $str \preceq str'_a \land str \preceq str'_b \Rightarrow$
$ssub(str'_a, str, ref) = ssub(str'_b, str)$
$\equiv str'_a = str'_b$

[CTA:ssub:subsub] $str_c \preceq str_a \land str_c \preceq str_b$
$\land str_b \preceq str_a$
$\Rightarrow ssub(ssub(str_a, str_c), ssub(str_b, str_c))$
$= ssub(str_a, str_b)$

[CTA:sadd:ssub] $str \preceq str' \Rightarrow$
$sadd(str, ssub(str', str)) = str'$

[CTA:ssub:sadd] $ssub(sadd(str_1, str_2), str_1) = str_2$

[CTA:SNC:sig] $ssync_{CTA} : \mathbb{P} E f E^* \times E^* f \mathbb{P}(E^*)$

[CTA:SNC:sym] $ssync(cs)(s_1, s_2) = ssync(cs)(s_2, s_1)$

[CTA:SNC:def]

$$
\begin{array}{rcl}
ssync(cs)(\langle\rangle, \langle\rangle) & \widehat{=} & \{ \langle\rangle \} \\
ssync(cs)(\langle\rangle, \langle a \rangle) & \widehat{=} & \emptyset \lhd a \in cs \rhd \{ \langle a \rangle \} \\
ssync(cs)(a{:}s, b{:}t) & \widehat{=} & \\
a \notin cs \land b \notin cs & \Rightarrow & (a{:})(ssync(cs)(s, b{:}t)) \cup (b{:})(ssync(cs)(a{:}s, t)) \\
a \notin cs \land b \in cs & \Rightarrow & (a{:})(ssync(cs)(s, b{:}t)) \\
a \in cs \land b \notin cs & \Rightarrow & (b{:})(ssync(cs)(a{:}s, t)) \\
a \in cs \land b \in cs & \Rightarrow & \emptyset \lhd a \neq b \rhd (a{:})(ssync(cs)(s, t))
\end{array}
$$

[CTA:SNC:one] $\forall s' \in ssync(cs)(s_1, hnull) \bullet acc(s') \subseteq acc(s_1) \setminus cs$

[CTA:SNC:only] $s' \in acc(ssync(cs)(s_1, s_2)) \Rightarrow acc(s') \subseteq acc(s_1) \cup acc(s_2)$

[CTA:SNC:sync] $s' \in acc(ssync(cs)(s_1, s_2)) \Rightarrow cs \cap acc(s') \subseteq cs \cap (acc(s_1) \cap acc(s_2))$

[CTA:SNC:assoc] $syncset(cs)(s_1)(ssync(cs)(s_2, s_3)) = syncset(cs)(s_3)(ssync(cs)(s_1, s_2))$

[CTA:SHid:sig] $shide_{CTA} : \mathbb{P} E f E^* f E^*$

[CTA:SHid:def] $shide(hdn)str \widehat{=} str \setminus hdn$

[CTA:SHid:evts] $acc(shide(hdn)str) = acc(str) \setminus hdn$

A.3 Slot, Slots level

A.3.1 Trace Equivalence of a Slot-Sequence

[ETs:sig] $EQVTRACE_{\mathscr{S}} : E^* \leftrightarrow (\mathscr{S}\ E)^+$

[ETs:def:nil] $EQVTRACE(tr, \langle\rangle) \mathrel{\widehat{=}} tr = \langle\rangle$

[ETs:def:cons] $EQVTRACE(tr, slot{:}slots) \mathrel{\widehat{=}} \exists tr_0 \bullet tr_0 \leq tr \wedge EQVTRC(tr_0, slot) \wedge EQVTRACE(tr - tr_0, slots)$

[ETs:sngl] $EQVTRACE(tr, \langle slot\rangle) \mathrel{\widehat{=}} EQVTRC(tr, slot)$

[ETs:cat] $EQVTRACE(tr_a, slots_a) \wedge EQVTRACE(tr_b, slots_b) \Rightarrow EQVTRACE(tr_a \frown tr_b, slots_a \frown slots_b)$

[ETs:elems] $EQVTRACE(tr, slots) \Rightarrow elems(tr) = \bigcup_{i \in 1}^{\#slots} \{ acc(slots(i)) \}$

[ETs:null] $EQVTRACE(\langle\rangle, slots) \equiv \forall i \in 1 \ldots \#slots \bullet EQVTRC(\langle\rangle, slots(i))$

[PEV:sig] $POSSEVTS : E^* \leftrightarrow (\mathscr{S}\ E)^+ \times (\mathscr{S}\ E)^+$

[PEV:def] $POSSEVTS(trace', (slots, slots')) \mathrel{\widehat{=}} \exists tr, tr' \bullet EQVTRACE(tr, slots) \wedge EQVTRACE(tr', slots') \wedge trace' = tr' - tr$

[SLOT:structure] $slot, (hist, ref) \in \mathscr{S}\ E \mathrel{\widehat{=}} \mathscr{H}\ E \times \mathbb{P} E$

[Overload:Sig:q:H] $q_{\mathscr{H}} : \mathscr{H}\ E f X$

[Overload:Sig:q:S] $q_{\mathscr{S}} : \mathscr{S}\ E f X$

[Overload:q:def] $q_{\mathscr{S}} = Q_{\mathscr{H}} \circ \pi_1$

[Overload:Sig:m:H] $m_{\mathscr{H}} : X f \mathscr{H}\ E f \mathscr{H}\ E$

[Overload:Sig:m:S] $m_{\mathscr{S}} : X f \mathscr{S}\ E f \mathscr{S}\ E$

[Overload:m:def] $m_{\mathscr{S}}(x)(hist, ref) \mathrel{\widehat{=}} (m_{\mathscr{H}}(x) hist, ref)$

A.3.2 Accepted and Refused Events and Equivalent Traces

[ACC:sig] $acc_{\mathscr{S}} : \mathscr{H}\ E f \mathbb{P} E$

$acc_{\mathscr{H}} : \mathscr{S}\ E f \mathbb{P} E$

[ET:sig] $EQVTRC_{\mathscr{H}} : E^* \leftrightarrow \mathscr{H}\ E$

$EQVTRC_{\mathscr{S}} : E^* \leftrightarrow \mathscr{S}\ E$

[ET:elems] ✠ $EQVTRC(tr, hist) \Rightarrow elems(tr) = acc(hist)$

[HIST:exists] ✠ $\exists hist \bullet acc(hist) = S$

$$[\text{REF:sig}] \qquad sref_{\mathscr{S}} : \mathscr{S}\ E \mathbb{f} \mathbb{P} E$$

$$[\text{sref:def}] \qquad sref \mathrel{\widehat=} \pi_2$$

$$[\text{HIST:eq}] \quad ✠ \quad (h_1 = h_2) \equiv \forall tr \bullet EQVTRC(tr, h_1) \equiv EQVTRC(tr, h_2)$$

$$[\text{SLOT:eq}] \qquad (s_1 = s_2) \equiv sref(s_1) = sref(s_2) \land \forall tr \bullet EQVTRC(tr, s_1) \equiv EQVTRC(tr, s_2)$$

$$[\text{Acc:h:eq:elems:ET}] \qquad acc(t) = S \equiv \forall r \exists tt \bullet EQVTRC(tt, (t, r)) \land elems(tt) = S$$

A.3.3 Null Slots

$$[\text{HN:sig}] \qquad hnull_{\mathscr{H}} : \mathscr{H}\ E$$

$$[\text{HN:null}] \quad ✠ \quad acc(hnull) = \{\,\}$$

$$[\text{SN:sig}] \qquad snull_{\mathscr{S}} : \mathbb{P} E \mathbb{f} \mathscr{S}\ E$$

$$[\text{SN:def}] \qquad snull(ref) \mathrel{\widehat=} (hnull, ref)$$

$$[\text{SN:ref}] \qquad sref(snull(ref)) = ref$$

$$[\text{SN:null}] \qquad acc(snull(ref)) = \{\,\}$$

$$[\text{SN:eq}] \qquad snull(r) = snull(r') \equiv r = r'$$

A.3.4 Slot Prefix Relation

$$[\text{pfx:sig}] \qquad \preceq_{\mathscr{H}} : \mathscr{H}\ E \leftrightarrow \mathscr{H}\ E$$

$$\preceq_{\mathscr{S}} : \mathscr{S}\ E \leftrightarrow \mathscr{S}\ E$$

$$[\text{pfx:def}] \qquad (hist_1, ref_1) \preceq_{\mathscr{S}} (hist_2, ref_2) \mathrel{\widehat=} hist_1 \preceq_{\mathscr{H}} hist_2$$

$$[\text{pfx:refl}] \quad ✠ \quad hist \preceq hist = \textsc{True}$$

$$slot \preceq slot = \textsc{True}$$

$$[\text{pfx:trans}] \quad ✠ \quad hist_1 \preceq hist_2 \land hist_2 \preceq hist_3 \Rightarrow hist_1 \preceq hist_3$$

$$slot_1 \preceq slot_2 \land slot_2 \preceq slot_3 \Rightarrow slot_1 \preceq slot_3$$

$$[\text{pfx:anti-sym}] \quad ✠ \quad hist_1 \preceq hist_2 \land hist_2 \preceq hist_1 \Rightarrow hist_1 = hist_2$$

[SN:pfx] ✠ $hnull \preceq hist$

$snull(r) \preceq slot$

[ET:pfx] ✠ $hist_1 \preceq hist_2 \Rightarrow \exists tr_1, tr_2 \bullet EQVTRC(tr_1, hist_1) \wedge EQVTRC(tr_2, hist_2) \wedge tr_1 \leq tr_2$

$slot_1 \preceq slot_2 \Rightarrow \exists tr_1, tr_2 \bullet EQVTRC(tr_1, s_1) \wedge EQVTRC(tr_2, s_2) \wedge tr_1 \leq tr_2$

[pfx:ignores:ref:1] $slot_1 \preceq slot_2 \wedge slot_2 \preceq slot_1 \not\Rightarrow sref(slot_1) = sref(slot_2)$

[pfx:ignores:ref:2] $slot_1 \preceq slot_2 \equiv \forall r_1, r_2 \bullet slot_1[r_1] \preceq slot_2[r_2]$

A.3.5 Slot Addition (Concatenation)

[sadd:sig] $sadd_{\mathscr{H}} : \mathscr{H}\ E \times \mathscr{H}\ E f \mathscr{H}\ E$

$sadd_{\mathscr{S}} : \mathscr{S}\ E \times \mathscr{S}\ E f \mathscr{S}\ E$

[sadd:def] $sadd_{\mathscr{S}}((h_1, r_1), (h_2, r_2)) \mathrel{\widehat{=}} (sadd_{\mathscr{H}}(h_1, h_2), r_2)$

[sadd:events] ✠ $acc(sadd(h_1, h_2)) = acc(h_1) \cup acc(h_2)$

$acc(sadd(s_1, s_2)) = acc(s_1) \cup acc(s_2)$

[sadd:ref] $sref(sadd(s_1, s_2)) = sref(s_2)$

[sadd:unit] ✠ $sadd(h_1, h_2) = h_1 \equiv h_2 = hnull$

✠ $sadd(h_1, h_2) = h_2 \equiv h_1 = Hnull$

$sadd(s_1, s_2) = s_1 \equiv s_2 = snull(sref(s_1))$

$sadd(s_1, s_2) = s_2 \equiv \exists r_2 \bullet s_2 = snull(r_2)$

[sadd:assoc] ✠ $sadd(h_1, sadd(h_2, h_3)) = sadd(sadd(h_1, h_2), h_3)$

$sadd(s_1, sadd(s_2, s_3)) = sadd(sadd(s_1, s_2), s_3)$

[sadd:prefix] ✠ $h \preceq sadd(h, h')$

$s \preceq sadd(s, s')$

[sadd:eqv:unit] $sadd(s_1, s_2) \approx s_1 \ \equiv \ \exists r_2 \bullet s_2 = snull(r_2)$

[sadd:binop] $s_1 \sharp s_2 \mathrel{\widehat{=}} sadd(s_1, s_2)$

A.3.6 Slot Subtraction

[ssub:sig] $ssub_{\mathscr{H}} : \mathscr{H}\ E \times \mathscr{H}\ E \to \mathscr{H}\ E$

$ssub_{\mathscr{S}} : \mathscr{S}\ E \times \mathscr{S}\ E \to \mathscr{S}\ E$

[ssub:def] $ssub_{\mathscr{S}}((h_1, r_1), (h_2, r_2)) \mathrel{\widehat{=}} (ssub_{\mathscr{H}}(h_1, h_2), r_1)$

[ssub:pre] ✠ $pre\text{-}ssub(h_1, h_2) = h_2 \preceq h_1$

$pre\text{-}ssub(s_1, s_2) = s_2 \preceq s_1$

[ssub:events] ✠ $h_2 \preceq h_1 \land h' = ssub(h_1, h_2) \Rightarrow$
$acc(h_1) \setminus acc(h_2) \subseteq acc(h') \subseteq acc(h_1)$

[ssub:events]$_{\mathscr{S}}$ $s_2 \preceq s_1 \land s' = ssub(s_1, s_2) \Rightarrow$
$acc(s_1) \setminus acc(s_2) \subseteq acc(s') \subseteq acc(s_1)$

[SSub:ref] $sref(ssub(slot', slot)) = sref(slot')$

[SSub:self] ✠ $ssub(h, h) = hnull$

$ssub(s, s) = snull(sref(s))$

[SSub:nil] ✠ $ssub(h, hnull) = h$

$ssub(s, snull(r)) = s$

[SSub:same] ✠ $hist \preceq hist'_a \land hist \preceq hist'_b \Rightarrow$
$ssub(hist'_a, hist) = ssub(hist'_b, hist) \equiv hist'_a = hist'_b$

[SSub:same]$_{\mathscr{S}}$ $slot \preceq slot'_a \land slot \preceq slot'_b \Rightarrow$
$ssub(slot'_a, slot) = ssub(slot'_b, slot) \equiv slot'_a = slot'_b$

[SSub:subsub] ✠ $hist_c \preceq hist_a \land hist_c \preceq hist_b \land hist_b \preceq hist_a$
$\Rightarrow ssub(ssub(hist_a, hist_c), ssub(hist_b, hist_c)) = ssub(hist_a, hist_b)$

[SSub:subsub]$_{\mathscr{S}}$ $slot_c \preceq slot_a \land slot_c \preceq slot_b \land slot_b \preceq slot_a$
$\Rightarrow ssub(ssub(slot_a, slot_c), ssub(slot_b, slot_c)) = ssub(slot_a, slot_b)$

[SSub:eqv] $s_1 \approx s_2 \equiv ssub(s_1, s_2) = snull(sref(s_1))$

[SSub:equal] $s = ssub(s, sn) \equiv \exists rn \bullet sn = snull(rn)$

[sadd:binop] $s_1 \setminus s_2 \mathrel{\widehat{=}} ssub(s_1, s_2)$

A.3.7 Relating Addition and Subtraction

$$
\begin{array}{rcl}
\text{[sadd:ssub]} & \Pisymbol{pzd}{66} & hist \preceq hist' \Rightarrow sadd(hist, ssub(hist', hist)) = hist' \\
 & & slot \preceq slot' \Rightarrow sadd(slot, ssub(slot', slot)) = slot' \\
\text{[ssub:sadd]} & \Pisymbol{pzd}{66} & ssub(sadd(h_1, h_2), h_1) = h_2 \\
 & & ssub(sadd(s_1, s_2), s_1) = s_2
\end{array}
$$

$$(S_1 \cup S_2) \setminus S_1 \neq S_2 \qquad \text{e.g.: } (\{a\} \cup \{a\}) \setminus \{a\} = \emptyset \neq \{a\}.$$

A.3.8 Hiding Slot Events

$$
\begin{array}{rl}
\text{[SHid:sig]} & SHide_{\mathscr{H}} : \mathbb{P}Ef\mathscr{H}\ Ef\mathscr{H}\ E \\
 & SHide_{\mathscr{S}} : \mathbb{P}Ef\mathscr{S}\ Ef\mathscr{S}\ E
\end{array}
$$

$$
\begin{array}{rcl}
\text{[SHid:def]} & & SHide_{\mathscr{S}}(hid)(hist, ref) \mathrel{\widehat{=}} (SHide_{\mathscr{H}}(hid)hist, ref) \\
\text{[SHid:evts]} & \Pisymbol{pzd}{66} & acc(SHide(hid)(h)) = acc(h) \setminus hid \\
 & & acc(SHide(hid)(s)) = acc(s) \setminus hid \\
\text{[SHid:refs]} & & sref(SHide(hid)(s)) = sref(s) \\
\text{[hide:it:is:null]} & & (\exists t \bullet acc(t) = \{c\} \wedge tt = SHide_H\{c\}(t)) \equiv acc(tt) = \emptyset
\end{array}
$$

A.3.9 Slot Synchronisation

$$
\begin{array}{rl}
\text{[SNC:sig]} & ssync_{\mathscr{H}} : \mathbb{P}Ef\mathscr{H}\ E \times \mathscr{H}\ Ef\mathbb{P}(\mathscr{H}\ E) \\
 & ssync_{\mathscr{S}} : \mathbb{P}Ef\mathscr{S}\ E \times \mathscr{S}\ Ef\mathbb{P}(\mathscr{S}\ E)
\end{array}
$$

$$
\begin{array}{rl}
\text{[SNC:def]} & ssync_{\mathscr{S}}(cs)((hist_1, ref_1), (hist_2, ref_2)) \\
 & \mathrel{\widehat{=}} ssync_{\mathscr{H}}(cs)(hist_1, hist_2) \times \{ rsync(cs)(ref_1, ref_2) \} \\
\text{[RSYN:sig]} & rsync : \mathbb{P}Ef\mathbb{P}E \times \mathbb{P}Ef\mathbb{P}E \\
\text{[RSYN:def]} & rsync(cs)(r_1, r_2) \mathrel{\widehat{=}} ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \setminus cs) \\
\text{[RSYN:sym]} & rsync(cs)(r_1, r_2) = rsync(cs)(r_2, r_1) \\
\text{[RSYN:assoc]} & rsync(cs)(r_1, rsync(cs)(r_2, r_3) = rsync(cs)(rsync(cs)(r_1, r_2), r_3)
\end{array}
$$

[SNC:sym] ✠ $ssync(cs)(h_1,h_2) = ssync(cs)(h_2,h_1)$

$ssync(cs)(s_1,s_2) = ssync(cs)(s_2,s_1)$

[SNC:null] $ssync(cs)(snull(r_1),snull(r_2)) = \{snull(rsync(r_1,r_2))\}$

[SNC:one] ✠ $\forall h' \in ssync(cs)(h_1,hnull) \bullet acc(h') \subseteq acc(h_1) \setminus cs$

$\forall r_2 \bullet \forall s' \in ssync(cs)(s_1,snull(r_2)) \bullet acc(s') \subseteq acc(s_1) \setminus cs$

[SNC:only] ✠ $h' \in acc(ssync(cs)(h_1,h_2)) \Rightarrow acc(h') \subseteq acc(h_1) \cup acc(h_2)$

$s' \in acc(ssync(cs)(s_1,s_2)) \Rightarrow acc(s') \subseteq acc(s_1) \cup acc(s_2)$

[SNC:sync] ✠ $h' \in acc(ssync(cs)(h_1,h_2)) \Rightarrow cs \cap acc(h') \subseteq cs \cap (acc(h_1) \cap acc(h_2))$

$s' \in acc(ssync(cs)(s_1,s_2)) \Rightarrow cs \cap acc(s') \subseteq cs \cap (acc(s_1) \cap acc(s_2))$

[SNCS:sig] $syncset : \mathbb{P} E f \mathscr{H}\, E f \mathbb{P}(\mathscr{H}\, E) f \mathbb{P}(\mathscr{H}\, E)$

[SNCS:def] $syncset(cs)(h)(H) \mathrel{\widehat=} \bigcup \{ ssync(cs)(h,h') \mid h' \in H \}$

[SNC:assoc] ✠ $syncset(cs)(h_1)(ssync(cs)(h_2,h_3)) = syncset(cs)(h_3)(ssync(cs)(h_1,h_2))$

$syncset(cs)(s_1)(ssync(cs)(s_2,s_3)) = syncset(cs)(s_3)(ssync(cs)(s_1,s_2))$

A.3.10 Parameter Summary

$$
\begin{aligned}
\mathscr{H} &: E f \mathscr{H}\, E \\
acc_{\mathscr{H}} &: \mathscr{H}\, E f \mathbb{P} E \\
EQVTRC_{\mathscr{H}} &: E^* \leftrightarrow \mathscr{H}\, E \\
hnull_{\mathscr{H}} &: \mathscr{H}\, E \\
\preceq_{\mathscr{H}} &: \mathscr{H}\, E \leftrightarrow \mathscr{H}\, E \\
ssub_{\mathscr{H}} &: \mathscr{H}\, E \times \mathscr{H}\, E \to \mathscr{H}\, E \\
sadd_{\mathscr{H}} &: \mathscr{H}\, E \times \mathscr{H}\, E f \mathscr{H}\, E \\
SHide_{\mathscr{H}} &: \mathbb{P} E f \mathscr{H}\, E f \mathscr{H}\, E \\
ssync_{\mathscr{H}} &: \mathbb{P} E f \mathscr{H}\, E \times \mathscr{H}\, E f \mathbb{P}(\mathscr{H}\, E)
\end{aligned}
$$

[ET:elems] ✠ $EQVTRC(tr,hist) \Rightarrow elems(tr) = acc(hist)$

[HIST:exists] ✠ $\exists hist \bullet acc(hist) = S$

[HIST:eq] ✠ $(h_1 = h_2) \equiv \forall tr \bullet EQVTRC(tr,h_1) \equiv EQVTRC(tr,h_2)$

[HN:null] ✠ $acc(hnull) = \{\}$

[pfx:refl] ✠ $hist \preceq hist = \text{TRUE}$

[pfx:trans] ✠ $hist_1 \preceq hist_2 \wedge hist_2 \preceq hist_3 \Rightarrow hist_1 \preceq hist_3$

[pfx:anti-sym] ✠ $hist_1 \preceq hist_2 \wedge hist_2 \preceq hist_1 \Rightarrow hist_1 = hist_2$

[SN:pfx] ✠ $hnull \preceq hist$

[ET:pfx] ✠ $hist_1 \preceq hist_2 \Rightarrow \exists tr_1, tr_2 \bullet EQVTRC(tr_1,hist_1) \wedge EQVTRC(tr_2,hist_2) \wedge tr_1 \leq tr_2$

(continued overleaf)

[sadd:events] ✠ $acc(sadd(h_1,h_2)) = acc(h_1) \cup acc(h_2)$

[sadd:unit:r] ✠ $sadd(h_1,h_2) = h_1 \equiv h_2 = hnull$

[sadd:unit:l] ✠ $sadd(h_1,h_2) = h_2 \equiv h_1 = hnull$

[sadd:assoc] ✠ $sadd(h_1,sadd(h_2,h_3)) = sadd(sadd(h_1,h_2),h_3)$

[sadd:prefix] ✠ $h \preceq sadd(h,h')$

[ssub:pre] ✠ $pre\text{-}ssub(h_1,h_2) = h_2 \preceq h_1$

[ssub:events] ✠ $h_2 \preceq h_1 \wedge h' = ssub(h_1,h_2) \Rightarrow$
$acc(h_1) \setminus acc(h_2) \subseteq acc(h') \subseteq acc(h_1)$

[SSub:self] ✠ $ssub(h,h) = hnull$

[SSub:nil] ✠ $ssub(h,hnull) = h$

[SSub:same] ✠ $hist \preceq hist'_a \wedge hist \preceq hist'_b \Rightarrow$
$ssub(hist'_a,hist) = ssub(hist'_b,hist) \equiv hist'_a = hist'_b$

[SSub:subsub] ✠ $hist_c \preceq hist_a \wedge hist_c \preceq hist_b \wedge hist_b \preceq hist_a$
$\Rightarrow ssub(ssub(hist_a,hist_c),ssub(hist_b,hist_c)) = ssub(hist_a,hist_b)$

[sadd:ssub] ✠ $hist \preceq hist' \Rightarrow sadd(hist,ssub(hist',hist)) = hist'$

[ssub:sadd] ✠ $ssub(sadd(h_1,h_2),h_1) = h_2$

[SHid:evts] ✠ $acc(SHide(hid)(h)) = acc(h) \setminus hid$

[SNC:sym] ✠ $ssync(cs)(h_1,h_2) = ssync(cs)(h_2,h_1)$

[SNC:one] ✠ $\forall h' \in ssync(cs)(h_1,hnull) \bullet acc(h') \subseteq acc(h_1) \setminus cs$

[SNC:only] ✠ $h' \in acc(ssync(cs)(h_1,h_2)) \Rightarrow acc(h') \subseteq acc(h_1) \cup acc(h_2)$

[SNC:sync] ✠ $h' \in acc(ssync(cs)(h_1,h_2)) \Rightarrow cs \cap acc(h') \subseteq cs \cap (acc(h_1) \cap acc(h_2))$

[SNC:assoc] ✠ $syncset(cs)(h_1)(ssync(cs)(h_2,h_3)) = syncset(cs)(h_3)(ssync(cs)(h_1,h_2))$

A.3.11 Extracting Refusal Sequences

[RFS:sig] $srefs_{\mathscr{S}} : (\mathscr{S}\ E)^+ f (\mathbb{P}E)^+$

[RFS:def] $srefs(slots) = map(sref)(slots)$

[ER:sig] $eqvref_{\mathscr{S}} : (\mathscr{S}\ E)^+ f \mathbb{P}E$

[ER:def] $eqvref(slots) \mathrel{\widehat{=}} sref(last(slots))$

A.3.12 Slot-Sequence Prefix Ordering ($\preccurlyeq$)

[EX:sig] $\preccurlyeq_{\mathscr{S}} : (\mathscr{S}\ E)^+ \leftrightarrow (\mathscr{S}\ E)^+$

[EX:def] $slots \preccurlyeq slots' \mathrel{\widehat{=}} front(slots) < slots' \wedge last(slots) \preceq slots'(\#slots)$

[EX:prior-hist-ref:def] $p^\frown\langle(h,r)\rangle \preccurlyeq p'^\frown\langle(h',r')\rangle$
$= p \le p' \wedge (h,r) \preceq (p'^\frown\langle(h',r')\rangle)(\#p+1)$

[EX:subseq] $slots_a \leq slots_b \Rightarrow slots_a \preccurlyeq slots_b$

[EX:refl] $slots \preccurlyeq slots$

[EX:trans] $slots_a \preccurlyeq slots_b \land slots_b \preccurlyeq slots_c \Rightarrow slots_a \preccurlyeq slots_c$

[EX:anti] $(\forall slot_a, slot_b \bullet slot_a \preceq slot_b \land slot_b \preceq slot_a \Rightarrow slot_a = slot_b)$
$\Rightarrow$
$(slots_a \preccurlyeq slots_b \land slots_b \preccurlyeq slots_a \Rightarrow slots_a = slots_b)$

[EX:null] $\langle snull(r) \rangle \preccurlyeq slots$

$$\begin{aligned} slots &= pfx^\frown \langle slot \rangle \\ slots' &= pfx^\frown \langle slot' \rangle^\frown sfx \end{aligned}$$
[EX:pfx]

[EX:prefix] $ss_1{}^\frown ss_2 \preccurlyeq ss_1{}^\frown ss_3 \equiv ss_2 \preccurlyeq ss_3$

[EX:sngl] $\langle s_1 \rangle \preccurlyeq s_2{:}ss \equiv s_1 \preceq s_2$

[EX;EX] $(slots \preccurlyeq slots'); (slots \preccurlyeq slots') = (slots \preccurlyeq slots')$

A.3.13 Slot Equivalences

[SEQV:sig] $\approx_{\mathscr{S}} : \mathscr{S}\ E \leftrightarrow \mathscr{S}\ E$

[SEQV:def] $slot_1 \approx slot_2 \mathrel{\widehat=} slot_1 \preceq slot_2 \land slot_2 \preceq slot_1$

[SEQV:equal-h] $(h_1, \_) \approx (h_2, \_) \equiv h_1 = h_2$

[SSEQV:sig] $\cong_{\mathscr{S}} : (\mathscr{S}\ E)^+ \leftrightarrow (\mathscr{S}\ E)^+$

[SSEQV:def] $slots_1 \cong slots_2 \mathrel{\widehat=} slots_1 \preccurlyeq slots_2 \land slots_2 \preccurlyeq slots_1$

[SEQV:refl] $slot \approx slot$

[SEQV:symm] $slot_1 \approx slot_2 \equiv slot_2 \approx slot_1$

[SEQV:trans] $slot_1 \approx slot_2 \wedge slot_2 \approx slot_3 \Rightarrow slot_1 \approx slot_3$

[SSEQV:refl] $slots \cong slots$

[SSEQV:symm] $slots_1 \cong slots_2 \equiv slots_2 \cong slots_1$

[SSEQV:trans] $slots_1 \cong slots_2 \wedge slots_2 \cong slots_3 \Rightarrow slots_1 \cong slots_3$

[SSEQV:expand] $slots_1 \cong slots_2$
$= front(slots_1) = front(slots_2) \wedge last(slots_1) \approx last(slots_2)$

[EX;SSEQV] $(slots \preccurlyeq slots'); (slots \cong slots') \equiv slots \preccurlyeq slots'$

[SSEQV;EX] $(slots \cong slots'); (slots \preccurlyeq slots') \equiv slots \preccurlyeq slots'$

[SSEQV;SSEQV] $(slots \cong slots'); (slots \cong slots') \equiv slots \cong slots'$

[sadd:eqv:unit] : $p.103$ $sadd(s_1, s_2) \approx s_1 \ \equiv ? \ \exists r \bullet s_2 = snull(r)$

[SSub:eqv] : $p.104$ $(\exists r \bullet ssub(s_1, s_2) = snull(r)) \ \equiv ? \ s_1 \approx s_2$

A.3.14 Slot-Sequence Addition

[CAT:sig] $\#\!\#_{\mathscr{S}} : ((\mathscr{S}\ E)^+ \times (\mathscr{S}\ E)^+) f (\mathscr{S}\ E)^+$

[CAT:def] $slots_1 \#\!\# slots_2 \mathrel{\widehat{=}} front(slots_1)^\frown \langle last(slots_1) \sharp head(slots_2) \rangle^\frown tail(slots_2)$

[CAT:assoc] $sl_1 \#\!\# (sl_2 \#\!\# sl_3) = (sl_1 \#\!\# sl_2) \#\!\# sl_3$

[CAT:PFX] $ss \preccurlyeq ss \#\!\# tt$

[CAT:ER:last] $eqvref(sl_1 \#\!\# sl_2) = eqvref(sl_2)$

[CAT:eqv] $sl_1 \cong sl_1 \#\!\# sl_2 \equiv \exists r_2 \bullet sl_2 = \langle snull(r_2) \rangle$

[CAT:equal] $sl_1 = sl_1 \#\!\# sl_2 \equiv sl_2 = \langle snull(eqvref(sl_1)) \rangle$

[CAT:len] $\#(sl_1 \#\!\# sl_2) = \#(sl_1) + \#(sl_2) - 1$

A.3.15 Slot-Sequence Subtraction

$$
\begin{array}{rl}
\text{[DF:sig]} & dif_{\mathscr{S}} : ((\mathscr{S}\ E)^+ \times (\mathscr{S}\ E)^+) \rightarrow (\mathscr{S}\ E)^+ \\
\text{[DF:pre]} & \textit{pre-}dif(slots', slots) = slots \preccurlyeq slots' \\
\text{[DF:def]} & dif(slots', slots) \mathrel{\widehat{=}} ssub(slot', slot) : sfx \\
\textbf{where} & slot = last(slots) \\
 & (slot' : sfx) = slots' - front(slots) \\
\text{[DF:pfx]} & dif(pfx^\frown \langle slot' \rangle^\frown sfx, pfx^\frown \langle slot \rangle) = ssub(slot', slot) : sfx
\end{array}
$$

$$
\begin{array}{rl}
\text{[DF:equal]} & slots = dif(slots, sln) \equiv \exists rn \bullet sln = \langle snull(rn) \rangle \\
\text{[DF:self]} & dif(slots, slots) = \langle snull(eqvref(slots)) \rangle \\
\text{[DF:nil]} & dif(slots, \langle snull(r) \rangle) = slots \\
\text{[DF:Null:equal]} & slots' \mathbin{\diagdown\!\diagdown} slots = \langle snull(r') \rangle \equiv slots' \cong slots \wedge eqvref(slots') = r' \\
\text{[DF:Null:eqv]} & (\exists r' \bullet slots' \mathbin{\diagdown\!\diagdown} slots \cong \langle snull(r') \rangle) \equiv slots' \cong slots \\
\text{[DF:same]} & slots_b \preccurlyeq slots_a \wedge slots_b \preccurlyeq slots_c \Rightarrow \\
 & \quad dif(slots_a, slots_b) = dif(slots_c, slots_b) \equiv slots_a = slots_c \\
\text{[DF:all-null]} & EQVTRACE(\langle\rangle, dif(slots_a, slots_b)) \equiv \\
 & \forall i \in 1 \ldots \#dif(slots_a, slots_b) \bullet EQVTRC(\langle\rangle, (dif(slots_a, slots_b))(i)) \\
\text{[DF:ref]} & srefs(dif(slots_a, slots_b)) = srefs(slots_a - front(slots_b)) \\
\text{[DF:hd-Evt]} & EQVTRACE(\langle\rangle, dif(slots', slots)) \Rightarrow EQVTRC(\langle\rangle, head(dif(slots', slots))) \\
\text{[DF:subsub]} & slots_c \preccurlyeq slots_a \wedge slots_c \preccurlyeq slots_b \wedge slots_b \preccurlyeq slots_a \\
 & \Rightarrow dif(dif(slots_a, slots_c), dif(slots_b, slots_c)) = dif(slots_a, slots_b) \\
\text{[EX:dif]} & slots \preccurlyeq slots' \equiv \langle snull(r) \rangle \preccurlyeq dif(slots', slots) \\
\text{[DF:ER:first]} & eqvref(sl_1 \mathbin{\diagdown\!\diagdown} sl_2) = eqvref(sl_1) \\
\text{[DF:len]} & \#(sl_1 \mathbin{\diagdown\!\diagdown} sl_2) = (\#(sl_1) - \#(sl_2)) + 1
\end{array}
$$

$$
\begin{array}{rll}
\text{[DF:binop]} & slots_1 \mathbin{\diagdown\!\diagdown} slots_2 \mathrel{\widehat{=}} dif(slots_1, slots_2) \\
 & = & \textbf{let}\ rest = slots_1 - front(slots_2) \\
 & & \textbf{in}\ (head(rest) \mathbin{\diagdown} last(slots_2)) : tail(rest)
\end{array}
$$

A.3.16 Relating Slot-Sequence Addition and Subtraction

$$
\begin{array}{rl}
\text{[CAT:DF:id]} & (ss \mathbin{\#\!\#} tt) \mathbin{\diagdown\!\diagdown} ss = tt \\
\text{[CAT:DF:pfx]} & (ss \mathbin{\#\!\#} tt) \mathbin{\diagdown\!\diagdown} (ss \mathbin{\#\!\#} uu) = tt \mathbin{\diagdown\!\diagdown} uu, \quad \textbf{if}\ uu \preccurlyeq tt
\end{array}
$$

A.4 Healthiness conditions

[R1:distr:and] $\mathbf{R1}(P \wedge Q) = \mathbf{R1}(P) \wedge \mathbf{R1}(Q) = \mathbf{R1}(P) \wedge Q = P \wedge \mathbf{R1}(Q)$

[R1:distr:or] $\mathbf{R1}(P \vee Q) = \mathbf{R1}(P) \vee \mathbf{R1}(Q)$

[R1:distr:all] $\mathbf{R1}(\forall x \bullet P) = \forall x \bullet \mathbf{R1}(P), x \notin \{ slots, slots' \}$

[R1:distr:any] $\mathbf{R1}(\exists x \bullet P) = \exists x \bullet \mathbf{R1}(P), x \notin \{ slots, slots' \}$

[R1:distr:cond] $\mathbf{R1}(P \lhd c \rhd Q) = \mathbf{R1}(P) \lhd c \rhd \mathbf{R1}(Q)$

[R2:distr:and] $\mathbf{R2}(P \wedge Q) \equiv \mathbf{R2}(P) \wedge Q, \quad slots, slots'$ not free in Q

[R2:distr:or] $\mathbf{R2}(P \vee Q) \equiv \mathbf{R2}(P) \vee \mathbf{R2}(Q)$

[R2:distr:cond] $\mathbf{R2}(P \lhd c \rhd Q) \equiv \mathbf{R2}(P) \lhd c \rhd \mathbf{R2}(Q), \quad slots, slots'$ not free in c

[R3:distr:and] $\mathbf{R3}(P \wedge Q) = \mathbf{R3}(P) \wedge \mathbf{R3}(Q)$

[R3:distr:or] $\mathbf{R3}(P \vee Q) = \mathbf{R3}(P) \vee \mathbf{R3}(Q)$

[R3:distr:cond] $\mathbf{R3}(P \lhd c \rhd Q) = \mathbf{R3}(P) \lhd c \rhd \mathbf{R3}(Q)$

[R3:wait:Skip] $(P \equiv \mathbf{R3}(P)) \Rightarrow wait \wedge P \equiv wait \wedge \mathit{II}_R$

[one-point:RSTET] $(\exists obs_0 \bullet P \wedge (RSTET[obs_0/obs])) \equiv (\exists ok_0, state_0 \bullet P[wait', slots'/wait_0, slots_0])$

[one-point:RSTET'] $(\exists obs_0 \bullet P \wedge (RSTET[obs_0/obs'])) \equiv (\exists ok_0, state_0 \bullet P[wait, slots/wait_0, slots_0])$

[comp:RSTET] $(P; Q \wedge RSTET)$
$\equiv \exists ok_0, state_0 \bullet P[ok_0, state_0/ok', state'] \wedge Q[ok_0, state_0, rest'/ok, state, rest]$

[comp:RSTET'] $(P \wedge RSTET; Q)$
$\equiv \exists ok_0, state_0 \bullet P[ok_0, state_0, rest/ok', state', rest'] \wedge Q[ok_0, state_0/ok, state]$

[DIV:DIV:eq:DIV] $DIV; DIV \equiv DIV$

[IIr:DIV:eq:DIV] $\mathit{II}_R; DIV \equiv DIV$

[DIV:IIr:eq:DIV] $DIV; \mathit{II}_R \equiv DIV$

[R1:is:R2] $\mathbf{R2}(\mathbf{R1}(\mathbf{true})) \equiv \mathbf{R1}(\mathbf{true})$

[IIr:is:R1] $\mathbf{R1}(\mathit{II}_R)) \equiv \mathit{II}_R$

[R3:is:R1] $\mathbf{R1}(P) \equiv P \Rightarrow \mathbf{R1}(\mathbf{R3}(P)) \equiv \mathbf{R3}(P)$

[IIr:is:R2] $\mathbf{R2}(\mathit{II}_R)) \equiv \mathit{II}_R$

[IIr:is:R3] $\mathbf{R3}(\mathit{II}_R)) \equiv \mathit{II}_R$

[R1:R2:comm] $\mathbf{R1} \circ \mathbf{R2} = \mathbf{R2} \circ \mathbf{R1}$

[R1:R3:comm] $\mathbf{R1} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{R1}$

[R2:R3:comm] $\mathbf{R2} \circ \mathbf{R3} = \mathbf{R3} \circ \mathbf{R2}$

[R:idem] $\mathbf{R} \circ \mathbf{R} = \mathbf{R}$

[R:distr:and] $\mathbf{R}(P \wedge Q) = \mathbf{R}(P) \wedge \mathbf{R}(Q),$
$(slots, slots'$ not free in $P) \vee (slots, slots'$ not free in $Q)$

[R:distr:or] $\mathbf{R}(P \vee Q) = \mathbf{R}(P) \vee \mathbf{R}(Q)$

[R:distr:cond] $\mathbf{R}(P \lhd c \rhd Q) = \mathbf{R}(P) \lhd c \rhd \mathbf{R}(Q), \quad slots, slots'$ not free in c

[DIV:is:R1] $\mathbf{R}1(DIV) = DIV$

[DIV:is:R2] $\mathbf{R}2(DIV) = DIV$

[Ilr:is:CSP1] $CSP1(\mathbb{I}_R) = \mathbb{I}_R$

[R1:CSP1:comm] $\mathbf{R}1 \circ CSP1 = CSP1 \circ \mathbf{R}1$

[R2:CSP1:comm] $\mathbf{R}2 \circ CSP1 = CSP1 \circ \mathbf{R}2$

[R3:CSP1:comm] $\mathbf{R}3 \circ CSP1 = CSP1 \circ \mathbf{R}3$

[CSP1:distr:and] $CSP1(P \wedge Q) = CSP1(P) \wedge CSP1(Q)$

[CSP1:distr:or] $CSP1(P \vee Q) = CSP1(P) \vee CSP1(Q)$

[CSP1:distr:cond] $CSP1(P \lhd c \rhd Q) = CSP1(P) \lhd c \rhd CSP1(Q)$

[CSP1:distr:all] $CSP1(\forall x \bullet P) = \forall x \bullet CSP1(P), x \notin \{ok, slots, slots'\}$

[CSP1:distr:any] $CSP1(\exists x \bullet P) = \exists x \bullet CSP1(P), x \notin \{ok, slots, slots'\}$

[comp:CSP1:closed] $(P \equiv CSP1(P)) \wedge (Q \equiv CSP1(Q)) \Rightarrow ((P; Q) \equiv CSP1(P; Q))$

[CSP2:distr:and] $CSP2(P \wedge Q) = CSP2(P) \wedge Q, \quad ok'$ not free in Q

[CSP2:distr:or] $CSP2(P \vee Q) = CSP2(P) \vee CSP2(Q)$

[CSP2:distr:any] $CSP2(\exists x \bullet P) = \exists x \bullet CSP2(P), \quad x \neq ok'$

[CSP2:distr:cond] $CSP2(P \lhd c \rhd Q) = CSP2(P) \lhd c \rhd CSP2(Q), \quad ok'$ not free in c

[Ilr:is:CSP2] $CSP2(\mathbb{I}_R) = \mathbb{I}_R$

[DIV:is:CSP2] $CSP2(DIV) = DIV$

[R1:CSP2:comm] $\mathbf{R}1 \circ CSP2 = CSP2 \circ \mathbf{R}1$

[R2:CSP2:comm] $\mathbf{R}2 \circ CSP2 = CSP2 \circ \mathbf{R}2$

[R3:CSP2:comm] $\mathbf{R}3 \circ CSP2 = CSP2 \circ \mathbf{R}3$

[CSP1:CSP2:comm] $CSP1 \circ CSP2 = CSP2 \circ CSP1$

[CSP3:distr:and] $CSP3(P \wedge Q) = CSP3(P) \wedge Q, \quad obs$ not free in Q

[CSP3:distr:or] $CSP3(P \vee Q) = CSP3(P) \vee CSP3(Q)$

[CSP3:distr:any] $CSP3(\exists x \bullet P) = \exists x \bullet CSP3(P), \quad x \notin obs$

[CSP3:distr:cond] $CSP3(P \lhd c \rhd Q) = CSP3(P) \lhd c \rhd CSP3(Q), \quad obs$ not free in c

[CSP4:distr:and] $CSP4(P \wedge Q) = CSP4(P) \wedge Q, \quad obs'$ not free in Q

[CSP4:distr:or] $CSP4(P \vee Q) = CSP4(P) \vee CSP4(Q)$

[CSP4:distr:any] $CSP4(\exists x \bullet P) = \exists x \bullet CSP4(P), \quad x \notin obs'$

[CSP4:distr:cond] $CSP4(P \lhd c \rhd Q) = CSP4(P) \lhd c \rhd CSP4(Q), \quad obs'$ not free in c

A.5 Basic Actions

[NEV:is:R1:R2] $\mathbf{R}2(\mathbf{R}1((NOEVTS)) \equiv NOEVTS$

[EVN:is:R1:R2] $\mathbf{R}2(\mathbf{R}1(EVTSNOW(E))) \equiv EVTSNOW(E)$

[IME:is:R1:R2] $\mathbf{R}2(\mathbf{R}1(IMMEVTS)) \equiv IMMEVTS$

[NEV:refl] $NOEVTS(ss, ss) \equiv \text{TRUE}$

[NEV:trans] $NOEVTS(ss, tt) \wedge NOEVTS(tt, uu) \Rightarrow NOEVTS(ss, uu)$

$NOEVTS(slots, slots'); NOEVTS(slots, slots') \equiv NOEVTS(slots, slots')$

[NEV:anti] $NOEVTS(ss, tt) \wedge NOEVTS(tt, ss) \Rightarrow ss \cong tt$

[EVN:irr] $EVTSNOW(E)(ss, ss) \equiv \text{FALSE} \qquad E \neq \emptyset$

[IME:irr] $IMMEVTS(ss, ss) \equiv \text{FALSE}$

Appendix B

Proofs

B.1 Basic Actions

B.1.1 BasicActionsLaw-1

[BasicActionsLaw-1] : $p.30$ $\quad EVTSNOW(\emptyset)(slots, slots') \equiv slots \cong slots'$

Proof:

$EVTSNOW(\emptyset)(slots, slots')$

$\equiv$ " [EVN:def]:p. 29 "

$\exists tt \bullet elems(tt) = \emptyset \wedge EQVTRACE(tt, slots' \setminus\!\!\setminus slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$

$\equiv$ " $tt = \langle\rangle$ "

$EQVTRACE(\langle\rangle, slots' \setminus\!\!\setminus slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$

$\equiv$ " slots are non empty sequences, $slots = pfx \frown \langle(t,r)\rangle$ "

$\exists t, r, pfx \bullet slots = pfx \frown \langle(t,r)\rangle$

$\wedge EQVTRACE(\langle\rangle, slots' \setminus\!\!\setminus slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$

$\equiv$ " [EX:pfx]:p. 108 "

$\exists t, r, pfx, t', r' \bullet slots = pfx \frown \langle(t,r)\rangle \wedge slots' = pfx \frown \langle(t',r')\rangle \frown sfx$

$\wedge EQVTRACE(\langle\rangle, slots' \setminus\!\!\setminus slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$

$\equiv$ " $\#slots = \#slots' \Rightarrow (sfx = \langle\rangle)$ "

$\exists t, r, pfx, t', r' \bullet slots = pfx \frown \langle(t,r)\rangle \wedge slots' = pfx \frown \langle(t',r')\rangle$

$\wedge EQVTRACE(\langle\rangle, slots' \setminus\!\!\setminus slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$

$\equiv$ " [DF:pfx]:p. 110 "

$\exists t, r, pfx, t', r' \bullet slots = pfx \frown \langle(t,r)\rangle \wedge slots' = pfx \frown \langle(t',r')\rangle$

$\wedge EQVTRACE(\langle\rangle, \langle(t',r')\rangle \setminus\!\!\setminus \langle(t,r)\rangle) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$

$\equiv$ " [ETs:null]:p. 101 "

$$\exists t,r,pfx,t',r' \bullet slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t',r') \rangle$$
$$\wedge EQVTRC(\langle\rangle, (t',r') \setminus (t,r)) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$$
$\equiv$ " Property of a history model "
$$\exists t,r,pfx,t',r' \bullet slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t',r') \rangle$$
$$\wedge t' = t \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$$
$\equiv$ " $t = t'$ "
$$\exists t,r,pfx,r' \bullet slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t,r') \rangle$$
$$\wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$$
$\equiv$ " [pfx:ignores:ref:1]:p. 103 "
$$\exists t,r,pfx,r' \bullet slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t,r') \rangle \wedge (t,r') \preceq (t,r)$$
$$\wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$$
$\equiv$ " [EX:def]:p. 107 "
$$slots' \preccurlyeq slots \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$$
$\equiv$ " [SSEQV:def]:p. 108 "
$$slots \cong slots'$$

B.1.2 BasicActionsLaw-3

[BasicActionsLaw-3] : $p.30$ $\quad \exists s,s' \bullet EVTSNOW\{c\}(s,s') \wedge sl' \nwarrow sl = map(SHide(\{c\}))(s' \nwarrow s)$
$\equiv sl' \cong sl$

Proof:

$$\exists s,s' \bullet EVTSNOW\{c\}(s,s') \wedge sl' \nwarrow sl = map(SHide(\{c\}))(s' \nwarrow s)$$
$\equiv$ " [EVN:def]:p. 29 "
$$\exists s,s',tt \bullet elems(tt) = \{c\} \wedge sl' \nwarrow sl = map(SHide(\{c\}))(s' \nwarrow s) \wedge$$
$$s \preccurlyeq s' \wedge \#s = \#s' \wedge EQVTRACE(tt, s' \nwarrow s)$$
$\equiv$ " [DF:len]:p. 110 "
$$\exists s,s',tt \bullet elems(tt) = \{c\} \wedge sl' \nwarrow sl = map(SHide(\{c\}))(s' \nwarrow s) \wedge$$
$$s \preccurlyeq s' \wedge \#(s' \nwarrow s) = 1 \wedge EQVTRACE(tt, s' \nwarrow s)$$
$\equiv$ " $s' \nwarrow s = \langle (t,r) \rangle$ "
$$\exists s,s',tt,t,r \bullet elems(tt) = \{c\} \wedge sl' \nwarrow sl = map(SHide(\{c\}))(\langle (t,r) \rangle) \wedge$$
$$s \preccurlyeq s' \wedge \#\langle (t,r) \rangle = 1 \wedge EQVTRACE(tt, \langle (t,r) \rangle) \wedge s' \nwarrow s = \langle (t,r) \rangle$$
$\equiv$ " Let $s = \langle SNull(r) \rangle$ and $s' = \langle (t,r) \rangle$ "
$$\exists tt,t,r \bullet elems(tt) = \{c\} \wedge sl' \nwarrow sl = map(SHide(\{c\}))(\langle (t,r) \rangle) \wedge$$
$$EQVTRACE(tt, \langle (t,r) \rangle)$$
$\equiv$ " [ETs:def:cons]:p. 101 "

$$\exists tt,t,r \bullet elems(tt) = \{c\} \wedge EQVTRC(tt,(t,r)) \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(\langle(t,r)\rangle)$$

$\equiv$ " [Acc:h:eq:elems:ET]:p. 102 "

$$\exists t,r \bullet acc(t) = \{c\} \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(\langle(t,r)\rangle)$$

$\equiv$ " *map* def "

$$\exists t,r \bullet acc(t) = \{c\} \wedge sl' \backslash\backslash sl = \langle SHide(\{c\})(t,r)\rangle$$

$\equiv$ " $sl' \backslash\backslash sl = \langle(tl,rl)\rangle$ "

$$\exists t,r,tl,rl \bullet acc(t) = \{c\} \wedge (tl,rl) = SHide(\{c\})(t,r) \wedge sl' \backslash\backslash sl = \langle(tl,rl)\rangle$$

$\equiv$ " [SHid:def]:p. 105 "

$$\exists t,tl,rl \bullet acc(t) = \{c\} \wedge tl = SHide_H\{c\}(t) \wedge sl' \backslash\backslash sl = \langle(tl,rl)\rangle$$

$\equiv$ " [hide:it:is:null]:p. 105 "

$$\exists tl,rl \bullet acc(tl) = \emptyset \wedge sl' \backslash\backslash sl = \langle(tl,rl)\rangle$$

$\equiv$ " [HN:null]:p. 106 "

$$\exists tl,rl \bullet tl = hnull \wedge sl' \backslash\backslash sl = \langle(tl,rl)\rangle$$

$\equiv$ " [SN:def]:p. 102 "

$$\exists rl \bullet sl' \backslash\backslash sl = \langle snull(rl)\rangle$$

$\equiv$ " [DF:Null:equal]:p. 110 "

$$\exists rl \bullet sl' \cong sl \wedge EQVREF(sl') = rl$$

$\equiv$ " One point rule "

$$sl' \cong sl$$

B.1.3 BasicActionsLaw-4

$$EVTSNOW(\emptyset)(slots,slots') \equiv slots \cong slots'$$

Proof "$\Rightarrow$"

$$EVTSNOW(\emptyset)(slots,slots')$$

$\equiv$ " [EVN:def]:p. 29 "

$$\exists tt \bullet elems(tt) = \emptyset \wedge EQVTRACE(tt,slots' \backslash\backslash slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$$

$\equiv$ " $tt = \langle\rangle$ "

$$EQVTRACE(\langle\rangle,slots' \backslash\backslash slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots'$$

$\equiv$ " slots are non empty sequences, $slots = pfx \frown \langle(t,r)\rangle$ "

$$
\begin{array}{ll}
& slots = pfx \frown \langle (t,r) \rangle \\
& \wedge EQVTRACE(\langle\rangle, slots' \setminus\!\setminus slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots' \\
\equiv & \quad \text{“ property of } slots \preccurlyeq slots' \text{ ”} \\
& slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t',r') \rangle \frown sfx \\
& \wedge EQVTRACE(\langle\rangle, slots' \setminus\!\setminus slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots' \\
\equiv & \quad \text{“ } \#slots = \#slots' \Rightarrow (sfx = \langle\rangle) \text{ ”} \\
& slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t',r') \rangle \\
& \wedge EQVTRACE(\langle\rangle, slots' \setminus\!\setminus slots) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots' \\
\equiv & \quad \text{“ property of } \setminus\!\setminus \text{ ”} \\
& slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t',r') \rangle \\
& \wedge EQVTRACE(\langle\rangle, \langle (t',r') \rangle \setminus\!\setminus \langle (t,r) \rangle) \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots' \\
\Rightarrow & \\
& slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t',r') \rangle \wedge t' = t \wedge \#slots = \#slots' \wedge slots \preccurlyeq slots' \\
\equiv & \\
& slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t',r') \rangle \wedge t' = t \wedge slots \preccurlyeq slots' \\
\Rightarrow & \text{“} t' = t \Rightarrow (t',r') \preceq (t,r) \text{”} \\
& slots = pfx \frown \langle (t,r) \rangle \wedge slots' = pfx \frown \langle (t',r') \rangle \wedge \langle (t',r') \rangle \preceq \langle (t,r) \rangle \wedge slots \preccurlyeq slots' \\
\Rightarrow & \text{“} \preccurlyeq def'' \\
& slots' \preccurlyeq slots \wedge slots \preccurlyeq slots' \\
\equiv & \quad \text{“} \cong \text{ definition ”} \\
& slots \cong slots'
\end{array}
$$

Proof “$\Leftarrow$”

$$
\begin{array}{ll}
& slots \cong slots' \\
\equiv & \quad \text{“} \cong \text{ definition ”} \\
& slots \cong slots' \wedge slots \preccurlyeq slots' \\
\equiv & \quad \text{“ property of } \cong \text{ ”} \\
& slots \cong slots' \wedge slots \preccurlyeq slots' \wedge \#slots = \#slots' \\
\equiv & \quad \text{“ property of } \setminus\!\setminus \text{ ”} \\
& slots \cong slots' \wedge slots \preccurlyeq slots' \wedge \#slots = \#slots' \wedge EqvTrc(\langle\rangle, slots' \setminus\!\setminus slots) \\
\Rightarrow & \\
& slots \preccurlyeq slots' \wedge \#slots = \#slots' \wedge EqvTrc(\langle\rangle, slots' \setminus\!\setminus slots) \\
\equiv & \quad \text{“ “EVTSNOW” definition ”} \\
& EVTSNOW(\emptyset)(slots, slots')
\end{array}
$$

B.2 Properties of Prefix

B.2.1 Non-terminating Prefix

[prefixLaw-1] $(c.e \rightarrow Skip) \wedge wait' \equiv CSP1(ok' \wedge \mathbf{R}(WTC(c))) \wedge wait'$

$$
\begin{array}{ll}
 & (c.e \rightarrow Skip) \wedge wait' \\
\equiv & \text{“ [Comm:def]:p. 34 ”} \\
 & CSP1\left(ok' \wedge \mathbf{R}3\left(WTC(c) \lhd wait' \rhd \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array}\right)\right)\right) \wedge wait' \\
\equiv & \text{“ [CSP1:def]:p. 26,[R3:def]:p. 26 ”} \\
 & \left(DIV \vee ok' \wedge I\!I_R \lhd wait \rhd \left(WTC(c) \lhd wait' \rhd \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array}\right)\right)\right) \wedge wait' \\
\equiv & \text{“ [Cond:def]:p. 33 ”} \\
 & \left(DIV \vee ok' \wedge I\!I_R \lhd wait \rhd \left(WTC(c) \wedge wait'\right)\right) \wedge wait' \\
\equiv & \text{“ [Cond:def]:p. 33 ”} \\
 & \left(DIV \vee ok' \wedge I\!I_R \lhd wait \rhd (WTC(c))\right) \wedge wait' \\
\equiv & \text{“ [CSP1:def]:p. 26,[R3:def]:p. 26 ”} \\
 & CSP1\left(ok' \wedge \mathbf{R}3\left(WTC(c)\right)\right) \wedge wait'
\end{array}
$$

B.2.2 Terminating prefix

$$\text{[prefixLaw-2]} \qquad (c.e \rightarrow Skip) \wedge \neg wait' \equiv CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array}\right)\right) \wedge \neg wait'$$

$$\begin{array}{ll} & (c.e \rightarrow Skip) \wedge \neg wait' \\ \equiv & \text{“ [Comm:def]:p. 34 ”} \\ & CSP1\left(ok' \wedge \mathbf{R}3\left(WTC(c) \lhd wait' \rhd \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array}\right)\right)\right) \wedge \neg wait' \\ \equiv & \text{“ [CSP1:def]:p. 26,[R3:def]:p. 26 ”} \\ & \left(DIV \vee ok' \wedge I\!I_R \lhd wait \rhd \left(WTC(c) \lhd wait' \rhd \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array}\right)\right)\right) \wedge \neg wait' \\ \equiv & \text{“ [Cond:def]:p. 33 ”} \\ & \left(DIV \vee ok' \wedge I\!I_R \lhd wait \rhd \left(state' = state \wedge WTC(c); EVTSNOW\{c\} \wedge \neg wait'\right)\right) \wedge \neg wait' \\ \equiv & \text{“ [Cond:def]:p. 33 ”} \\ & \left(DIV \vee ok' \wedge I\!I_R \lhd wait \rhd \left(state' = state \wedge WTC(c); EVTSNOW\{c\}\right)\right) \wedge \neg wait' \\ \equiv & \text{“ [CSP1:def]:p. 26,[R3:def]:p. 26 ”} \\ & CSP1\left(ok' \wedge \mathbf{R}3\left(state' = state \wedge WTC(c); EVTSNOW\{c\}\right)\right) \wedge \neg wait' \end{array}$$

B.2.3 Idle Prefix

Given healthy P:

$$
\begin{array}{ll}
\text{[prefixLaw-3]} & (c.e \rightarrow P) \wedge NOEVTS(slots, slots') \\
& \equiv \qquad \text{for healthy } P \\
& CSP1(ok' \wedge \mathbf{R}3(WTC(c) \wedge wait')) \wedge NOEVTS(slots, slots')
\end{array}
$$

Proof:

$$
\begin{array}{ll}
& (c.e \rightarrow P) \wedge NOEVTS(slots, slots') \\
\equiv & \text{“ [Pfx:def]:p. 34 ”} \\
& ((c.e \rightarrow Skip); P) \wedge wait' \wedge NOEVTS(slots, slots') \\
\equiv & \text{“ [Comm:def]:p. 34 ”} \\
& \left(CSP1\left(ok' \wedge \mathbf{R}3\left(WTC(c) \lhd wait' \rhd \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array} \right) \right) \right); P \right) \\
& \quad \wedge NOEVTS(slots, slots') \\
\equiv & \text{“ property of NOEVTS and seq comp ”} \\
& \left(CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} WTC(c) \\ \lhd wait' \rhd \\ \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array} \right) \end{array} \right) \right) \wedge NOEVTS(slots, slots'); P \right) \right. \\
& \quad \left. \wedge NOEVTS(slots, slots') \right) \\
\equiv & \text{“ Logic ”} \\
& \left(CSP1\left(ok' \wedge \mathbf{R}3\left(\left(\begin{array}{l} WTC(c) \\ \lhd wait' \rhd \\ \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array} \right) \end{array} \right) \wedge NOEVTS(slots, slots') \right) \right); P \right) \right. \\
& \quad \left. \wedge NOEVTS(slots, slots') \right) \\
\equiv & \text{“ Logic ”} \\
& \left(CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} WTC(c) \wedge NOEVTS(slots, slots') \\ \lhd wait' \rhd \\ \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array} \right) \wedge NOEVTS(slots, slots') \end{array} \right) \right); P \right) \right. \\
& \quad \left. \wedge NOEVTS(slots, slots') \right) \\
\equiv & \text{“ } EVTSNOW\{c\} \wedge NOEVENTS = False \text{ ”} \\
& \left(CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} WTC(c) \wedge NOEVTS(slots, slots') \\ \wedge wait' \end{array} \right) \right); P \right) \\
& \quad \wedge NOEVTS(slots, slots') \\
\equiv & \text{“ P is R3 ”} \\
& \left(CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} WTC(c) \wedge NOEVTS(slots, slots') \\ \wedge wait' \end{array} \right) \right); \mathbb{I}_R \right) \wedge NOEVTS(slots, slots') \\
\equiv & \text{“ Property of } \mathbb{I}_R \text{ ”} \\
& CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} WTC(c) \wedge wait' \end{array} \right) \right) \wedge NOEVTS(slots, slots')
\end{array}
$$

Restricting Prefix Refusals

$$WTC(c);EVTSNOW\{c\} \land c \in \bigcap srefs(slots' \setminus\setminus slots)$$
$$\equiv$$
$$EVTSNOW\{c\} \land c \in \bigcap srefs(slots' \setminus\setminus slots)$$

Proof

$$WTC(c);EVTSNOW\{c\} \land c \in \bigcap srefs(slots' \setminus\setminus slots)$$
$$\equiv \quad \text{“ [WTC:def]:p. 34 ”}$$
$$(NOEVTS(slots, slots') \land c \notin \bigcup srefs(slots' \setminus\setminus slots))$$
$$;EVTSNOW\{c\}(slots, slots')$$
$$\land c \in \bigcap srefs(slots' \setminus\setminus slots)$$
$$\equiv \quad \text{“ [Seq:def]:p. 33 ”}$$
$$\exists slots_0 \bullet$$
$$NOEVTS(slots, slots_0) \land c \notin \bigcup srefs(slots_0 \setminus\setminus slots)$$
$$\land EVTSNOW\{c\}(slots_0, slots')$$
$$\land c \in \bigcap srefs(slots' \setminus\setminus slots)$$
$$\equiv \quad \text{“ case split ”}$$
$$\#slots = \#slots' \land \exists slots_0 \bullet$$
$$NOEVTS(slots, slots_0) \land c \notin \bigcup srefs(slots_0 \setminus\setminus slots)$$
$$\land EVTSNOW\{c\}(slots_0, slots')$$
$$\land c \in \bigcap srefs(slots' \setminus\setminus slots)$$
$$\lor \#slots \neq \#slots' \land \exists slots_0 \bullet$$
$$NOEVTS(slots, slots_0) \land c \notin \bigcup srefs(slots_0 \setminus\setminus slots)$$
$$\land EVTSNOW\{c\}(slots_0, slots')$$
$$\land c \in \bigcap srefs(slots' \setminus\setminus slots)$$
$$\equiv \quad \text{“ case1 ”}$$

$$
\begin{aligned}
\equiv \quad & \text{“ case1 ”} \\
& EVTSNOW\{c\} \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
& \vee \#slots \neq \#slots' \wedge \exists slots_0 \bullet \\
& NOEVTS(slots, slots_0) \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
& \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ case2 ”} \\
& EVTSNOW\{c\} \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
& \vee false \\
\equiv \quad & \text{“ logic ”} \\
& EVTSNOW\{c\} \wedge c \in \bigcap srefs(slots' \setminus\setminus slots)
\end{aligned}
$$

Case1

$$
\begin{array}{ll}
 & \#slots = \#slots' \wedge \exists slots_0 \bullet \\
 & NOEVTS(slots, slots_0) \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ [NEV:is:R1:R2]:p. 113, [EVN:is:R1:R2]:p. 113, arithmetic ”} \\
 & \exists slots_0 \bullet \\
 & NOEVTS(slots, slots_0) \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
 & \wedge \#slots_0 = \#slots' \wedge \#slots = \#slots_0 \\
\equiv & \quad \text{“ [BasicActionsLaw-1]:p. 30, [BasicActionsLaw-3]:p. 30 ”} \\
 & \exists slots_0 \bullet \\
 & c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
 & \wedge slots \cong slots_0 \wedge \#slots = \#slots_0 \\
\equiv & \quad \text{“ property of } \setminus\setminus \text{ ”} \\
 & \exists slots_0 \bullet \\
 & c \notin sref(last(slots_0)) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
 & \wedge slots \cong slots_0 \wedge \#slots = \#slots_0 \\
\equiv & \quad \text{“ [EVN:def]:p. 29 ”} \\
 & \exists slots_0, tt \bullet \\
 & c \notin sref(last(slots_0)) \\
 & \wedge elems(tt) = \{c.e\} \wedge EQVTRACE(tt, slots' \setminus\setminus slots_0) \wedge \#slots' = \#slots_0 \\
 & \wedge slots \cong slots_0 \wedge \#slots = \#slots_0 \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ [property of slots equivalence]:p. } \mathbf{??} \text{ ”} \\
 & \exists slots_0, tt \bullet \\
 & c \notin sref(last(slots_0)) \\
 & \wedge elems(tt) = \{c.e\} \wedge EQVTRACE(tt, slots' \setminus\setminus slots) \wedge \#slots' = \#slots_0 \\
 & \wedge slots \cong slots_0 \wedge \#slots = \#slots_0 \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ arithmetics ”}
\end{array}
$$

$\equiv$ " arithmetics "

$\exists slots_0, tt \bullet$

$c \notin sref(last(slots_0))$

$\wedge elems(tt) = \{c.e\} \wedge EQVTRACE(tt, slots' \setminus\setminus slots) \wedge \#slots' = \#slots$

$\wedge slots \cong slots_0 \wedge \#slots = \#slots_0$

$\wedge c \in \bigcap srefs(slots' \setminus\setminus slots)$

$\equiv$ " [EVN:def]:p. 29 "

$\exists slots_0 \bullet$

$c \notin sref(last(slots_0))$

$EVTSNOW\{c\}(slots, slots')$

$\wedge slots \cong slots_0 \wedge \#slots = \#slots_0$

$\wedge c \in \bigcap srefs(slots' \setminus\setminus slots)$

$\equiv$ " [SSEQV:expand]:p. 109, [pfx:ignores:ref:2]:p. 103, [SEQV:def]:p. 108 "

$EVTSNOW\{c\}(slots, slots')$

$\wedge c \in \bigcap srefs(slots' \setminus\setminus slots)$

Case2

$$
\begin{array}{ll}
 & \#slots \neq \#slots' \wedge \exists slots_0 \bullet \\
 & NOEVTS(slots, slots_0) \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ [EVN:def]:p. 29 ”} \\
 & \#slots \neq \#slots' \wedge \exists slots_0 \bullet \\
 & NOEVTS(slots, slots_0) \wedge \#slots \leq \#slots_0 \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \wedge \#slots_0 = \#slots' \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ [DF:len]:p. 110,arithmetics ”} \\
 & \exists slots_0 \bullet \\
 & NOEVTS(slots, slots_0) \wedge \#(slots_0 \setminus\setminus slots) > 1 \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \wedge \#slots_0 = \#slots' \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ } slots_0 \setminus\setminus slots \text{ has more then one slot ”} \\
 & \exists slots_0, t, r, sl, s \bullet \\
 & NOEVTS(slots, slots_0) \wedge \#(slots_0 \setminus\setminus slots) > 1 \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \wedge \#slots_0 = \#slots' \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
 & \wedge slots_0 \setminus\setminus slots = \langle (t,r) \rangle \frown sl \frown \langle s \rangle \\
\equiv & \quad \text{“ set theory ”} \\
 & \exists slots_0, t, r, sl, s \bullet \\
 & NOEVTS(slots, slots_0) \wedge \#(slots_0 \setminus\setminus slots) > 1 \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \wedge \#slots_0 = \#slots' \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
 & \wedge slots_0 \setminus\setminus slots = \langle (t,r) \rangle \frown sl \frown \langle s \rangle \wedge c \notin Ref(t,r) \\
\equiv & \quad \text{“ [EVN:is:R1:R2]:p. 113 ”} \\
 & \exists slots_0, t, r, sl, s \bullet \\
 & NOEVTS(slots, slots_0) \wedge \#(slots_0 \setminus\setminus slots) > 1 \wedge c \notin \bigcup srefs(slots_0 \setminus\setminus slots) \\
 & \wedge EVTSNOW\{c\}(slots_0, slots') \wedge \#slots_0 = \#slots' \wedge slots_0 \preccurlyeq slots' \\
 & \wedge c \in \bigcap srefs(slots' \setminus\setminus slots) \\
 & \wedge slots_0 \setminus\setminus slots = \langle (t,r) \rangle \frown sl \frown \langle s \rangle \wedge c \notin Ref(t,r) \\
\Rightarrow &
\end{array}
$$

$$
\begin{array}{ll}
\Rightarrow & \\
 & \exists slots_0, t, r, sl, s \bullet \\
 & \wedge slots_0 \preccurlyeq slots' \\
 & \wedge slots_0 \setminus\!\setminus slots = \langle (t,r) \rangle \frown sl \frown \langle s \rangle \wedge c \notin Ref(t,r) \\
 & \wedge c \in \bigcap srefs(slots' \setminus\!\setminus slots) \\
\Rightarrow & \quad \textit{“properties of slot substraction and ordering'} \\
 & \exists slots_0, t, r, sl, s \bullet \\
 & \langle (t,r) \rangle \frown sl \leq slots' \setminus\!\setminus slots \wedge c \notin Ref(t,r) \\
 & \wedge c \in \bigcap srefs(slots' \setminus\!\setminus slots) \\
\Rightarrow & \quad \textit{“set theory, srefs def”} \\
 & \exists slots_0, t, r, sl, s \bullet \\
 & \langle (t,r) \rangle \frown sl \leq slots' \setminus\!\setminus slots \wedge c \notin Ref(t,r) \\
 & \wedge c \in \bigcap srefs(slots' \setminus\!\setminus slots) \wedge c \in Ref(t,r) \\
\equiv & \quad \text{“ Logic ”} \\
 & False
\end{array}
$$

B.3 Laws of Hiding

B.3.1 Hiding Prefix

$$(c.e \rightarrow Skip)\backslash\{c\} \equiv Skip\backslash\{c\}$$

$$
\begin{array}{ll}
 & (c.e \rightarrow Skip)\backslash\{c\} \\
\equiv & \text{“ Prefix definition ”} \\
 & (CSP1(ok' \wedge \mathbf{R}(WTC(c) \lhd wait' \rhd CMPC(c.e))))\backslash\{c\} \\
\equiv & \text{“ hiding def ”} \\
 & (\forall e_1 \bullet CSP1(ok' \wedge \mathbf{R}(\\
 & WTC(c) \wedge \{c.e_1\} \subset \bigcap Refs(slots' \setminus\setminus slots) \wedge wait' \\
 & \vee CMPC(c.e) \wedge \{c.e_1\} \subset \bigcap Refs(slots' \setminus\setminus slots) \wedge \neg wait' \\
 &)))\backslash\{c\} \\
\equiv & \text{“ } WTC, POSS \text{ def ”} \\
 & (\forall e_1, e_2 \bullet CSP1(ok' \wedge \mathbf{R}(\\
 & WTC(c) \wedge c.e_2 \notin \bigcup Refs(slots' \setminus\setminus slots) \wedge \{c.e_1\} \subset \bigcap Refs(slots' \setminus\setminus slots) \wedge wait' \\
 & \vee CMPC(c.e) \wedge \{c.e_1\} \subset \bigcap Refs(slots' \setminus\setminus slots) \wedge \neg wait' \\
 &)))\backslash\{c\} \\
\equiv & \text{“ set theory ”} \\
 & (\forall e_1 \bullet CSP1(ok' \wedge \mathbf{R}(\\
 & False \\
 & \vee CMPC(c.e) \wedge c.e_1 \in \bigcap Refs(slots' \setminus\setminus slots) \wedge \neg wait' \\
 &)))\backslash\{c\} \\
\equiv & \text{“ Lemma 7 ”} \\
 & (\forall e_1 \bullet CSP1(ok' \wedge \mathbf{R}(state = state' \wedge \\
 & TRMC(c.e) \wedge c.e_1 \in \bigcap Refs(slots' \setminus\setminus slots) \wedge \neg wait' \\
 &)))\backslash\{c\} \\
\equiv & \text{“ hiding def ”} \\
 & (CSP1(ok' \wedge \mathbf{R}(state = state' \wedge \\
 & TRMC(c.e) \wedge \neg wait' \\
 &)))\backslash\{c\} \\
\equiv & \text{“ } TRMC \text{ def ”} \\
 & (CSP1(ok' \wedge \mathbf{R}(state = state' \wedge \\
 & EVTSNOW(c.e)(slots, slots') \wedge \neg wait' \\
 &)))\backslash\{c\} \\
\equiv & \text{“ hiding def ”}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \text{“ hiding def ”} \\
& (\exists s' \bullet CSP1(ok' \wedge \mathbf{R}(state = state' \wedge \\
& EVTSNOW(c.e)(slots, s') \wedge \neg wait' \wedge slots' \setminus\setminus slots = map(SHide(\{c\}.e))(s' \setminus\setminus slots) \\
&))) \setminus \{c\} \\
\equiv & \text{“ Lemma 6 ”} \\
& (CSP1(ok' \wedge \mathbf{R}(state = state' \wedge \\
& \neg wait' \wedge slots' \cong slots \\
&))) \setminus \{c\} \\
\equiv & \text{“ Lemma 8 ”} \\
& (\mathbf{R}(CSP1(ok' \wedge state = state' \wedge \\
& \neg wait' \wedge slots' \cong slots \\
&))) \setminus \{c\} \\
\equiv & \text{“ } \mathbf{R}3 \text{ def ”} \\
& (\mathbf{R}(CSP1(ok' \wedge state = state' \wedge \\
& wait = wait' \wedge slots' \cong slots \\
&))) \setminus \{c\} \\
\equiv & \text{“ } CSP1 \text{ def ”} \\
& (\mathbf{R}(DIV \vee ok' \wedge state = state' \wedge \\
& wait = wait' \wedge slots' \cong slots \\
&)) \setminus \{c\} \\
\equiv & \text{“ Logic ”} \\
& (\mathbf{R}(\exists slots_r \bullet DIV \vee ok' \wedge state = state' \wedge \\
& wait = wait' \wedge slots' \cong slots \wedge slots' = slots_r \\
&)) \setminus \{c\} \\
\equiv & \text{“ Logic ”} \\
& (\mathbf{R}(\exists slots_r \bullet DIV \vee ok' \wedge state = state' \wedge \\
& wait = wait' \wedge slots_r \cong slots \wedge slots' = slots_r \\
&)) \setminus \{c\} \\
\equiv & \text{“ } I\!I \text{ def and } \preccurlyeq \text{ property ”} \\
& (\mathbf{R}(\exists slots_r \bullet slots_r \cong slots \wedge I\!I[slots_r/slots] \\
&)) \setminus \{c\} \\
\equiv & \text{“ } Skip \text{ def ”} \\
& (Skip) \setminus \{c\}
\end{array}
$$

B.3.2 Hiding Skip

$$(Skip) \setminus \{c\} = Skip$$

Proof:

$$
\begin{array}{ll}
 & (Skip) \setminus \{c\} \\
\equiv & \text{“ hiding def ”} \\
 & \mathbf{R}\left(\begin{array}{l} \forall e \exists s' \bullet Skip[s'/slots'] \wedge \\ slots' \setminus slots = map(SHide(hidn.e))(s' \setminus slots) \wedge \\ hidn.e \subset \bigcap Refs(s' \setminus slots) \end{array}\right); Skip \\
\equiv & \text{“ } .e \text{ def ”} \\
 & \mathbf{R}\left(\begin{array}{l} \forall e \exists s' \bullet Skip[s'/slots'] \wedge \\ slots' \setminus slots = map(SHide(hidn.e))(s' \setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus slots) \end{array}\right); Skip \\
\equiv & \text{“ set theory ”} \\
 & \mathbf{R}\left(\begin{array}{l} \forall e \exists s' \bullet Skip[s'/slots'] \wedge \\ slots' \setminus slots = map(SHide(hidn.e))(s' \setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus slots) \end{array}\right); Skip \\
\equiv & \text{“ Skip def alt ”} \\
 & \mathbf{R}\left(\begin{array}{l} \forall e \exists s' \bullet \\ \mathbf{R}3(CSP1(state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots'))[s'/slots'] \wedge \\ slots' \setminus slots = map(SHide(hidn.e))(s' \setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus slots) \end{array}\right); Skip \\
\equiv & \text{“ Lemma 10 ”} \\
 & \mathbf{R}\left(\begin{array}{l} \forall e \exists s' \bullet \\ CSP1(state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots')[s'/slots'] \wedge \\ slots' \setminus slots = map(SHide(hidn.e))(s' \setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus slots) \end{array}\right); Skip \\
\equiv & \text{“ } CSP1 \text{ def ”} \\
 & \mathbf{R}\left(\begin{array}{l} \forall e \exists s' \bullet \\ ((\neg ok \wedge slots \preccurlyeq slots') \vee (state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots'))[s'/slots'] \wedge \\ slots' \setminus slots = map(SHide(hidn.e))(s' \setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus slots) \end{array}\right); Skip \\
\equiv & \text{“ case split ”}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{R}\left(\begin{array}{l}\forall e \exists s' \bullet \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong s' \wedge \\ slots' \setminus\setminus slots = map(SHide(hidn.e))(s' \setminus\setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus\setminus slots)\end{array}\right); Skip \\
\vee & \\
 & \mathbf{R}\left(\begin{array}{l}\forall e \exists s' \bullet \\ \neg ok \wedge slots \preccurlyeq s' \wedge \\ slots' \setminus\setminus slots = map(SHide(hidn.e))(s' \setminus\setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus\setminus slots)\end{array}\right); Skip \\
\equiv & \text{“ skip is } CSP1 \text{ ”} \\
 & \mathbf{R}\left(\begin{array}{l}\forall e \exists s' \bullet \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong s' \wedge \\ slots' \setminus\setminus slots = map(SHide(hidn.e))(s' \setminus\setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus\setminus slots)\end{array}\right); Skip \\
\vee & \\
 & DIV \\
\equiv & \text{“ } CSP1 \text{ def ”} \\
 & CSP1(\mathbf{R}\left(\begin{array}{l}\forall e \exists s' \bullet \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong s' \wedge \\ slots' \setminus\setminus slots = map(SHide(hidn.e))(s' \setminus\setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus\setminus slots)\end{array}\right); Skip) \\
\equiv & \text{“ } head(Refs(slots')) = r \text{ ”} \\
 & CSP1(\mathbf{R}\left(\begin{array}{l}\forall e \exists s', r \bullet head(Refs(slots')) = r \wedge \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong s' \wedge \\ slots' \setminus\setminus slots = map(SHide(hidn.e))(s' \setminus\setminus slots) \wedge \\ c.e \in \bigcap Refs(s' \setminus\setminus slots)\end{array}\right); Skip) \\
\equiv & \text{“ } \setminus\setminus \text{ property ”} \\
 & CSP1(\mathbf{R}\left(\begin{array}{l}\forall e \exists s', r \bullet head(Refs(slots')) = r \wedge \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong s' \wedge \\ slots' \setminus\setminus slots = map(SHide(hidn.e))(\langle SNull(r)\rangle) \wedge \\ c.e \in \bigcap Refs(\langle SNull(r)\rangle)\end{array}\right); Skip) \\
\equiv & \text{“ } map \text{ and } SHide \text{ def ”} \\
 & CSP1(\mathbf{R}\left(\begin{array}{l}\forall e \exists s', r \bullet head(Refs(slots')) = r \wedge \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong s' \wedge \\ slots' \setminus\setminus slots = \langle SNull(r)\rangle \wedge \\ c.e \in \bigcap Refs(\langle SNull(r)\rangle)\end{array}\right); Skip) \\
\equiv & \text{“ one point rule ”} \\
 & CSP1(\mathbf{R}\left(\begin{array}{l}\forall e \exists r \bullet head(Refs(slots')) = r \wedge \\ state = state' \wedge \neg wait' \wedge ok' \wedge \\ slots' \setminus\setminus slots = \langle SNull(r)\rangle \wedge \\ c.e \in \bigcap Refs(\langle SNull(r)\rangle)\end{array}\right); Skip) \\
\equiv & \text{“ } \setminus\setminus \text{ property, } Refs \text{ def ”}
\end{array}
$$

$$CSP1(\mathbf{R}\left(\begin{array}{l}\forall e \exists r \bullet head(Refs(slots')) = r \wedge \\ state = state' \wedge \neg wait' \wedge ok' \wedge \\ slots' \cong slots \wedge \\ c.e \in r\end{array}\right);Skip)$$

$\equiv$ " logic "

$$CSP1(\mathbf{R}\left(\begin{array}{l}\forall e \bullet \\ state = state' \wedge \neg wait' \wedge ok' \wedge \\ slots' \cong slots \wedge \\ c.e \in head(Refs(slots'))\end{array}\right);Skip)$$

$\equiv$ " Skip is healthy "

$$CSP1(\mathbf{R}(\left(\begin{array}{l}\forall e \bullet \\ state = state' \wedge \neg wait' \wedge ok' \wedge \\ slots' \cong slots \wedge \\ c.e \in head(Refs(slots'))\end{array}\right);Skip))$$

$\equiv$ " Skip and seq com def "

$$CSP1(\mathbf{R}(\exists obs_0 \bullet \left(\begin{array}{l}\forall e \bullet \\ state = state_0 \wedge \neg wait_0 \wedge ok_0 \wedge \\ slots_0 \cong slots \wedge \\ c.e \in head(Refs(slots_0))\end{array}\right) \wedge$$
$$\mathbf{R}3(CSP1(state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots'))[obs_0/obs]))$$

$\equiv$ " $\neg wait_0 \wedge ok_0$ "

$$CSP1(\mathbf{R}(\exists obs_0 \bullet \left(\begin{array}{l}\forall e \bullet \\ state = state_0 \wedge \neg wait_0 \wedge ok_0 \wedge \\ slots_0 \cong slots \wedge \\ c.e \in head(Refs(slots_0))\end{array}\right) \wedge$$
$$(state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots')[obs_0/obs]))$$

$\equiv$ " renaming "

$$CSP1(\mathbf{R}(\exists obs_0 \bullet \left(\begin{array}{l}\forall e \bullet \\ state = state_0 \wedge \neg wait_0 \wedge ok_0 \wedge \\ slots_0 \cong slots \wedge \\ c.e \in head(Refs(slots_0)) \\ state_0 = state' \wedge \neg wait' \wedge ok' \wedge slots_0 \cong slots'\end{array}\right)))$$

$\equiv$ " logic "

$$CSP1(\mathbf{R}(\exists obs_0 \bullet \left(\begin{array}{l}\forall e \bullet \\ state = state_0 \wedge \neg wait_0 \wedge ok_0 \wedge \\ slots_0 \cong slots \wedge \\ c.e \in head(Refs(slots_0)) \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots'\end{array}\right)))$$

$\equiv$ " one point rule "

$$CSP1(\mathbf{R}(state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots'))$$

$\equiv$ " Skip def alt "$Skip$

B.3.3 Lemmas

Lemma2

$$slots_a \preccurlyeq slots_c \wedge slots_b \preccurlyeq slots_c \Rightarrow$$
$$((dif(slots_c, slots_a) = dif(slots_c, slots_b)) \equiv (slots_a \cong slots_b))$$

Lemma4

$$\forall c \exists t \bullet elems(t) = \{c\} \wedge tt = SHide_H\{c\}(t) \equiv elems(tt) = \emptyset$$

Lemma5

$$elems(t) = S \equiv \forall r \exists tt \bullet EqvTrc(tt, \langle t, r\rangle) \wedge elems(tt) = S$$

Lemma 6

$$\exists s, s' \bullet EVTSNOW\{c\}(s, s') \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(s' \backslash\backslash s) \equiv sl' \cong sl$$

Proof

$\exists s, s' \bullet EVTSNOW\{c\}(s, s') \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(s' \backslash\backslash s)$

$\equiv$ " $EVTSNOW$ def "

$\exists s, s', tt \bullet elems(tt) = \{c\} \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(s' \backslash\backslash s) \wedge$
$s \preccurlyeq s' \wedge \#s = \#s' \wedge EqvTrace(tt, s' \backslash\backslash s)$

$\equiv$ " property of $\backslash\backslash$ "

$\exists s, s', tt \bullet elems(tt) = \{c\} \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(s' \backslash\backslash s) \wedge$
$s \preccurlyeq s' \wedge \#(s' \backslash\backslash s) = 1 \wedge EqvTrace(tt, s' \backslash\backslash s)$

$\equiv$ " $s' \backslash\backslash s = \langle (t, r)\rangle$ "

$\exists s, s', tt, t, r \bullet elems(tt) = \{c\} \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(\langle (t, r)\rangle) \wedge$
$s \preccurlyeq s' \wedge \#\langle (t, r)\rangle = 1 \wedge EqvTrace(tt, \langle (t, r)\rangle) \wedge s' \backslash\backslash s = \langle (t, r)\rangle$

$\equiv$ " Let $s = \langle SNull(r)\rangle$ and $s' = \langle (t, r)\rangle$ "

$\exists tt, t, r \bullet elems(tt) = \{c\} \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(\langle (t, r)\rangle) \wedge$
$EqvTrace(tt, \langle (t, r)\rangle)$

$\equiv$ " $EqvTrace$ def "

$\exists tt, t, r \bullet elems(tt) = \{c\} \wedge EqvTrc(tt, (t, r)) \wedge sl' \backslash\backslash sl = map(SHide(\{c\}))(\langle (t, r)\rangle)$

$\equiv$ " Lemma5 "

$$
\begin{array}{ll}
 & \exists t, r \bullet elems(t) = \{c\} \land sl' \backslash\backslash sl = map(SHide(\{c\}))(\langle (t,r) \rangle) \\
\equiv & \text{“ } map \text{ def ”} \\
 & \exists t, r \bullet elems(t) = \{c\} \land sl' \backslash\backslash sl = \langle SHide(\{c\})(t,r) \rangle \\
\equiv & \text{“ } sl' \backslash\backslash sl = \langle (tl,rl) \rangle \text{ ”} \\
 & \exists t, r, tl, rl \bullet elems(t) = \{c\} \land (tl,rl) = SHide(\{c\})(\langle t,r \rangle) \land sl' \backslash\backslash sl = \langle (tl,rl) \rangle \\
\equiv & \text{“ } SHide \text{ def ”} \\
 & \exists t, tl, rl \bullet elems(t) = \{c\} \land tl = SHide_H\{c\}(t) \land sl' \backslash\backslash sl = \langle (tl,rl) \rangle \\
\equiv & \text{“ Lemma 4 ”} \\
 & \exists tl, rl \bullet elems(tl) = \emptyset \land sl' \backslash\backslash sl = \langle (tl,rl) \rangle \\
\equiv & \text{“ } SNull \text{ def ”} \\
 & \exists rl \bullet sl' \backslash\backslash sl = \langle SNull(rl) \rangle \\
\equiv & \text{“ [DF:Null:equal] ”} \\
 & \exists rl \bullet sl' \cong sl \land EqvRef(sl') = rl \\
\equiv & \text{“ One point rule ”} \\
 & sl' \cong sl
\end{array}
$$

Lemma7

$$
\begin{array}{l}
WTC(c); EVTSNOW\{c\} \land c_1 \in \bigcap Refs(slots' \backslash\backslash slots) \\
\equiv \\
EVTSNOW\{c\} \land c_1 \in \bigcap Refs(slots' \backslash\backslash slots)
\end{array}
$$

Proof

$$
\begin{array}{ll}
 & (WTC(c); (EVTSNOW\{c\}(slots, slots')) \\
 & \land c_1 \in \bigcap Refs(slots' \backslash\backslash slots) \\
\equiv & \text{“ WTC def, [EVN:sig]:p. 29 ”} \\
 & (POSS(c) \land NoEvents(slots, slots')); (EVTSNOW\{c\}(slots, slots') \\
 & \land \#slots = \#slots') \\
 & \land c_1 \in \bigcap Refs(slots' \backslash\backslash slots) \\
\equiv & \text{“ POSS def ”} \\
 & (\forall e_0 \bullet c_0 \notin \bigcup Refs(slots' \backslash\backslash slots) \land NoEvents(slots, slots')); (EVTSNOW\{c\}(slots, slots') \\
 & \land \#slots = \#slots') \\
 & \land c_1 \in \bigcap Refs(slots' \backslash\backslash slots) \\
\equiv & \text{“ seq composition def ”}
\end{array}
$$

$$
\begin{array}{ll}
 & \forall e_0 \exists slots_0 \bullet \\
 & c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge \#slots_0 = \#slots' \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ } NoEvents \text{ is GROW ”} \\
 & \forall e_0 \exists slots_0 \bullet \\
 & c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge \#slots_0 = \#slots' \wedge \#slots \leq \#slots_0 \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ case split ”} \\
 & \#slots = \#slots' \wedge \\
 & \forall e_0 \exists slots_0 \bullet \\
 & c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge \#slots_0 = \#slots' \wedge \#slots \leq \#slots_0 \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
 & \vee \#slots \neq \#slots' \wedge \\
 & \forall e_0 \exists slots_0 \bullet \\
 & c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge \#slots_0 = \#slots' \wedge \#slots \leq \#slots_0 \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ case1 ”} \\
 & EVTSNOW\{c\} \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
 & \vee \#slots \neq \#slots' \wedge \\
 & \forall e_0 \exists slots_0 \bullet \\
 & c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
 & \wedge \#slots_0 = \#slots' \wedge \#slots \leq \#slots_0 \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv & \quad \text{“ case2 ”} \\
 & EVTSNOW\{c\} \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
 & \vee False \\
\equiv & \quad \text{“ logic ”} \\
 & EVTSNOW\{c\} \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots)
\end{array}
$$

Case1

$$
\begin{aligned}
& \#slots = \#slots' \wedge \\
& \forall e_0 \exists slots_0 \bullet \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \wedge \#slots \leq \#slots_0 \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ Arithmetics ”} \\
& \#slots = \#slots' \wedge \\
& \forall e_0 \exists slots_0 \bullet \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \wedge \#slots = \#slots_0 \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ Lema1, Lema3 ”} \\
& \#slots = \#slots' \wedge \\
& \forall e_0 \exists slots_0 \bullet \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge slots \cong slots_0 \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ property of } \setminus\setminus \text{ ”} \\
& \forall e_0 \exists slots_0 \bullet \#slots = \#slot' \wedge \\
& c_0 \notin Ref(last(slots_0)) \wedge slots \cong slots_0 \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ } EVTSNOW \text{ def ”} \\
& \forall e_0 \exists slots_0 \bullet \#slots = \#slot' \wedge \\
& c_0 \notin Ref(last(slots_0)) \wedge slots \cong slots_0 \\
& \wedge \exists tt \bullet elems(tt) = \{c\} \wedge EqvTrace(tt, slots' \setminus\setminus slots_0) \wedge \#slots_0 = \#slots' \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ Lema2 ”} \\
& \forall e_0 \exists slots_0 \bullet \#slots = \#slot' \wedge \\
& c_0 \notin Ref(last(slots_0)) \wedge slots \cong slots_0 \\
& \wedge \exists tt \bullet elems(tt) = \{c\} \wedge EqvTrace(tt, slots' \setminus\setminus slots) \wedge \#slots_0 = \#slots' \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ reordering and arithmetics ”} \\
& \forall e_0 \exists slots_0 \bullet \\
& c_0 \notin Ref(last(slots_0)) \wedge slots \cong slots_0 \wedge \#slots_0 = \#slots \\
& \wedge \exists tt \bullet elems(tt) = \{c\} \wedge EqvTrace(tt, slots' \setminus\setminus slots) \wedge \#slots = \#slot' \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ } EVTSNOW \text{ def ”}
\end{aligned}
$$

$$
\begin{array}{ll}
\equiv & \text{“ } EVTSNOW \text{ def ”} \\
& \forall e_0 \exists slots_0 \bullet \\
& c_0 \notin Ref(last(slots_0)) \wedge slots \cong slots_0 \wedge \#slots_0 = \#slots \\
& \wedge EVTSNOW\{c\}(slots, slots') \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv & \text{“ property of } \cong \text{ ”} \\
& EVTSNOW\{c\}(slots, slots') \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv & \text{“ } TRMC \text{ def ”} \\
& EVTSNOW\{c\} \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots)
\end{array}
$$

Case2

$$
\begin{aligned}
& \#slots \neq \#slots' \wedge \\
& \forall e_0 \exists slots_0 \bullet \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \wedge \#slots \leq \#slots_0 \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ arithmetics ”} \\
& \forall e_0 \exists slots_0 \bullet \#slots < \#slot' \wedge \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \wedge \#slots < \#slots_0 \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ property of } \setminus\setminus \text{ ”} \\
& \forall e_0 \exists slots_0 \bullet \#slots < \#slot' \wedge \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \wedge \#slots < \#slots_0 \wedge \#(slots_0 \setminus\setminus slots) > 1 \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ } slots_0 \setminus\setminus slots \text{ has more then one slot ”} \\
& \forall e_0 \exists slots_0, t, r, sl, s \bullet \#slots < \#slot' \wedge \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \wedge \#slots < \#slots_0 \wedge \#(slots_0 \setminus\setminus slots) > 1 \\
& \wedge slots_0 \setminus\setminus slots = \langle (t,r) \rangle \frown sl \frown \langle s \rangle \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ set theory ”} \\
& \forall e_0 \exists slots_0, t, r, sl, s \bullet \#slots < \#slot' \wedge \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \\
& \wedge \#slots_0 = \#slots' \wedge \#slots < \#slots_0 \wedge \#(slots_0 \setminus\setminus slots) > 1 \\
& \wedge slots_0 \setminus\setminus slots = \langle (t,r) \rangle \frown sl \frown \langle s \rangle \wedge c_0 \notin Ref(t,r) \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\equiv \quad & \text{“ } EVTSNOW \text{ is } \mathbf{R}1 \text{ ”} \\
& \forall e_0 \exists slots_0, t, r, sl, s \bullet \#slots < \#slot' \wedge \\
& c_0 \notin \bigcup Refs(slots_0 \setminus\setminus slots) \wedge NoEvents(slots, slots_0) \wedge EVTSNOW\{c\}(slots_0, slots') \wedge slots_0 \preccurlyeq slots' \\
& \wedge \#slots_0 = \#slots' \wedge \#slots < \#slots_0 \wedge \#(slots_0 \setminus\setminus slots) > 1 \\
& \wedge slots_0 \setminus\setminus slots = \langle (t,r) \rangle \frown sl \frown \langle s \rangle \wedge c_0 \notin Ref(t,r) \\
& \wedge c_1 \in \bigcap Refs(slots' \setminus\setminus slots) \\
\Rightarrow &
\end{aligned}
$$

$$
\begin{array}{ll}
\Rightarrow & \\
 & \forall e_0 \exists slots_0, t, r, sl, s \bullet \\
 & \wedge slots_0 \preccurlyeq slots' \\
 & \wedge slots_0 \setminus\!\setminus slots = \langle (t,r) \rangle \frown sl \frown \langle s \rangle \wedge c_0 \notin Ref(t,r) \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\!\setminus slots) \\
\Rightarrow & \quad \text{“}\setminus\!\setminus \, def\text{”} \\
 & \forall e_0 \exists slots_0, t, r, sl, s \bullet \\
 & \wedge slots_0 \preccurlyeq slots' \\
 & \wedge slots_0 = front(slots) \frown \langle (t,r) \rangle \frown sl \frown \langle s \rangle \wedge c_0 \notin Ref(t,r) \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\!\setminus slots) \\
\Rightarrow & \quad \text{“}\preccurlyeq \, def\text{”} \\
 & \forall e_0 \exists t, r \bullet \\
 & \wedge front(slots) \frown \langle (t,r) \rangle < slots' \\
 & \wedge c_0 \notin Ref(t,r) \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\!\setminus slots) \\
\Rightarrow & \quad \text{“}\setminus\!\setminus \, def\text{”} \\
 & \forall e_0 \exists t, r \bullet \\
 & \wedge \langle Ssub((t,r), last(slots)) \rangle < slots' \setminus\!\setminus slots \\
 & \wedge c_0 \notin Ref(t,r) \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\!\setminus slots) \\
\Rightarrow & \quad \text{“}Ssub\ def\text{”} \\
 & \forall e_0 \exists t, t2, r \bullet \\
 & \wedge \langle (t2,r) \rangle < slots' \setminus\!\setminus slots \\
 & \wedge c_0 \notin Ref(t,r) \\
 & \wedge c_1 \in \bigcap Refs(slots' \setminus\!\setminus slots) \\
\Rightarrow & \quad \text{“}\bigcap \, and\ Refs\ def\text{”} \\
 & \forall e_0 \exists t, t2, r \bullet \\
 & \wedge \langle (t2,r) \rangle < slots' \setminus\!\setminus slots \\
 & \wedge c_0 \notin Ref(t,r) \\
 & \wedge c_1 \in Ref(t2,r) \\
\equiv & \quad \text{“}\, Ref \text{ def}\, \text{”} \\
 & False
\end{array}
$$

Lemma 8

$CSP1(ok' \wedge \mathbf{R}(P)) \equiv \mathbf{R}(CSP1(ok' \wedge P))$

Lemma 9

$CSP1(P) \equiv CSP1(ok' \wedge P)$

Lemma 10

$\mathbf{R}3(P) \equiv \mathbf{R}3(wait' \wedge P)$

B.4 Laws of Prioritised Choice

B.4.1 R2 commutes with PRI

[R2:PRI:comm] $\quad \mathbf{PRI}(\mathbf{R}2(P)) \equiv \mathbf{R}2(\mathbf{PRI}(P))$

$\mathbf{R}2(\mathbf{PRI}(P))$

$\equiv$ " [PRI:def]:p. 57 "

$\mathbf{R}2(P \wedge (ok \Rightarrow head(srefs(slots' \setminus\setminus slots)) \subseteq last(srefs(slots))))$

$\equiv$ " [R2:def]:p. 26 "

$\exists ss \bullet$

$(P \wedge (ok \Rightarrow head(srefs(slots' \setminus\setminus slots)) \subseteq eqvref(slots)))[ss, ss \# (slots' \setminus\setminus slots)/slots, slots']$

$\wedge eqvref(ss) = eqvref(slots)$

$\equiv$ " renaming "

$\exists ss \bullet$

$P[ss, ss \# (slots' \setminus\setminus slots)/slots, slots']$

$\wedge (ok \Rightarrow head(srefs((ss \# (slots' \setminus\setminus slots)) \setminus\setminus ss)) \subseteq eqvref(ss))$

$\wedge eqvref(ss) = eqvref(slots)$

$\equiv$ " [CAT:DF:id]:p. 110 "

$\exists ss \bullet$

$P[ss, ss \# (slots' \setminus\setminus slots)/slots, slots']$

$\wedge (ok \Rightarrow head(srefs(slots' \setminus\setminus slots)) \subseteq eqvref(ss))$

$\wedge eqvref(ss) = eqvref(slots)$

$\equiv$ " logic "

$\exists ss \bullet$

$P[ss, ss \# (slots' \setminus\setminus slots)/slots, slots']$

$\wedge (ok \Rightarrow head(srefs(slots' \setminus\setminus slots)) \subseteq eqvref(slots))$

$\wedge eqvref(ss) = eqvref(slots)$

$\equiv$ " logic "

$$
\begin{array}{ll}
\equiv & \text{“ logic ”} \\
& (ok \Rightarrow head(srefs(slots' \lessdot slots)) \subseteq eqvref(slots)) \\
& \wedge \exists ss \bullet \\
& P[ss, ss \sharp (slots' \lessdot slots)/slots, slots'] \\
& \wedge eqvref(ss) = eqvref(slots) \\
\equiv & \text{“ [R2:def]:p. 26 ”} \\
& (ok \Rightarrow head(srefs(slots' \lessdot slots)) \subseteq eqvref(slots)) \\
& \wedge \mathbf{R}2(P) \\
\equiv & \text{“ [PRI:def]:p. 57 ”} \\
& \mathbf{PRI}(\mathbf{R}2(P))
\end{array}
$$

B.4.2 Hiding Prioritised Choice

$$[\text{hiding:PriChoice}] \qquad (a \rightarrow P \overleftarrow{\Box} Q) \setminus \{a\} \equiv P \setminus \{a\}$$

$$
\begin{array}{ll}
& (a \rightarrow P \overleftarrow{\Box} Q) \setminus \{a\} \\
\equiv & \text{“ [pExt:def]:p. 56 ”} \\
& \left(\begin{array}{l} a \rightarrow P \wedge Q \wedge Stop \\ \vee Choice(a \rightarrow P, Q) \\ \vee WeakChoice(Q, a \rightarrow P) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ hiding distributes over disjunction ”} \\
& (a \rightarrow P \wedge Q \wedge Stop) \setminus \{a\} \\
& \vee Choice(a \rightarrow P, Q) \setminus \{a\} \\
& \vee WeakChoice(Q, a \rightarrow P) \setminus \{a\} \\
\equiv & \text{“ Split into 3 cases ”} \\
& CSP1(\mathbf{R}3(False)) \vee \\
& P \setminus \{a\} \vee \\
& CSP1(\mathbf{R}3(False)) \\
\equiv & \text{“ P is CSP1 and R3, hiding preserves healthiness ”} \\
& CSP1(\mathbf{R}3(False)) \vee \\
& CSP1(\mathbf{R}3(P \setminus \{a\})) \vee \\
& CSP1(\mathbf{R}3(False)) \\
\equiv & \text{“ Properties of CSP1 and R3 ”} \\
& P \setminus \{a\}
\end{array}
$$

Case 1

One process wants to communicate. Stop does not allow him to do that and makes him wait - $wait' - True$, while hiding makes an event c internal and forces the communication to perform it. Both can only be satisfied by the *miracle*

[CommAndStop:reduce] $(c \to P \wedge Q \wedge Stop) \setminus \{c\} \equiv CSP1(\mathbf{R}3(False))$

$$
\begin{array}{ll}
 & (c \to P \wedge Q \wedge Stop) \setminus \{c\} \\
\equiv & \text{“ [Stop:def]:p. 31 ”} \\
 & \left(\begin{array}{l} c \to P \wedge Q \\ \wedge CSP1(\mathbf{R}3(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ [Pfx:def]:p. 34 ”} \\
 & \left(\begin{array}{l} (c \to Skip; P) \wedge Q \\ \wedge CSP1(\mathbf{R}3(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ [Comm:def]:p. 34 ”} \\
 & \left(\begin{array}{l} CSP1\left(ok' \wedge \mathbf{R}3\left(WTC(c) \lhd wait' \rhd \left(\begin{array}{l} state' = state \wedge \\ WTC(c); EVTSNOW\{c\} \end{array} \right) \right) \right); P \\ \wedge Q \\ \wedge CSP1(\mathbf{R}3(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ Lemma 1, properties of R3 and CSP1 ”} \\
 & \left(\begin{array}{l} CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} \left(WTC(c) \wedge NOEVTS \right) \\ \lhd wait' \rhd \\ \left(\begin{array}{l} state' = state \wedge NOEVTS \wedge \\ WTC(c); EVTSNOW\{c\} \end{array} \right) \end{array} \right) \right); P \\ \wedge Q \\ \wedge CSP1(\mathbf{R}3(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ [Hid:def]:p. 39 ”} \\
 & \left(\begin{array}{l} CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} \left(WTC(c) \wedge NOEVTS \right) \\ \lhd wait' \rhd \\ \left(\begin{array}{l} state' = state \wedge NOEVTS \wedge \\ WTC(c); EVTSNOW\{c\} \end{array} \right) \end{array} \right) \right); P \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R}3(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ Lemma 1 ”} \\
 & \left(\begin{array}{l} CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} \left(WTC(c) \wedge NOEVTS \right) \\ \lhd wait' \rhd \\ \left(\begin{array}{l} state' = state \wedge NOEVTS \wedge \\ \left(\begin{array}{l} (WTC(c) \wedge NOEVTS); \\ (EVTSNOW\{c\} \wedge NOEVTS) \end{array} \right) \end{array} \right) \end{array} \right) \right); P \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R}3(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ Lemma 2 ”} \\
 & \left(\begin{array}{l} CSP1\left(ok' \wedge \mathbf{R}3\left(\begin{array}{l} \left(WTC(c) \wedge NOEVTS \right) \\ \lhd wait' \rhd \\ \left(\begin{array}{l} state' = state \wedge NOEVTS \wedge \\ \left((WTC(c) \wedge NOEVTS); False \right) \end{array} \right) \end{array} \right) \right); P \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R}3(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ [Seq:def]:p. 33 ”}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \text{“ [Seq:def]:p. 33 ”} \\
& \left(\begin{array}{l} CSP1\left(ok' \wedge \mathbf{R3}\left(\begin{array}{l} \left(\begin{array}{l} WTC(c) \wedge NOEVTS \end{array} \right) \\ \lhd wait' \rhd \\ False \end{array} \right) \right) ; P \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R3}(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ [Cond:def]:p. 33 ”} \\
& \left(\begin{array}{l} \left(CSP1\left(ok' \wedge \mathbf{R3}\left(WTC(c) \wedge NOEVTS \wedge \neg wait' \right) \right) \right) ; P \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R3}(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ [WTC:def]:p. 34 ”} \\
& \left(\begin{array}{l} \left(\begin{array}{l} CSP1\left(ok' \wedge \mathbf{R3}\left(\begin{array}{l} c \notin \bigcup srefs(slots' \setminus\setminus slots) \wedge \\ NOEVTS \wedge \neg wait' \end{array} \right) \right) ; P \end{array} \right) \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R3}(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ Lemma 3, P is PRI healthy ”} \\
& \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} CSP1\left(ok' \wedge \mathbf{R3}\left(\begin{array}{l} c \notin \bigcup srefs(slots' \setminus\setminus slots) \wedge \\ NOEVTS \wedge \neg wait' \end{array} \right) \right) \\ \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \end{array} \right) \\ ; P \end{array} \right) \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R3}(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ [CSP1:def]:p. 26, [R3:def]:p. 26 ”} \\
& \left(\begin{array}{l} \left(\left(CSP1(\mathbf{R3}(False)) \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \right) ; P \right) \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R3}(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ P is CSP1 and R3 ”} \\
& \left(\begin{array}{l} CSP1(\mathbf{R3}(False)) \\ \wedge Q \wedge \{c\} \subseteq \bigcap srefs(slots' \setminus\setminus slots) \\ \wedge CSP1(\mathbf{R3}(ok' \wedge wait' \wedge NOEVTS)) \end{array} \right) \setminus \{c\} \\
\equiv & \text{“ [Hid:def]:p. 39 ”} \\
& CSP1(\mathbf{R3}(False))
\end{array}
$$

Case 2

In here we have to assume that Q is prefix healthy. Then if no time travels are allowed and Q is not a miracle then

$$[\text{NullPrefix:Assumption}] \exists obs' \bullet Q \wedge ok \wedge \neg wait \wedge slots \cong slots'$$

If the high priority event is chosen, then it will behave like the high priority event. This part of the proof is the same like in case of external choice. Because an event will be performed immediately, there is no problem of the second process not agreeing on waiting.

[Choice:perform] $Choice(a \to P, Q) \setminus \{a\} \equiv P \setminus \{a\}$

$$
\begin{array}{ll}
 & Choice(a \to P, Q) \setminus \{a\} \\
\equiv & \text{“ [Choice:def]:p. 36 ”} \\
 & CSP2 \left(\begin{array}{l} a \to P \wedge \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ [Hid:def]:p. 39 ”} \\
 & CSP2 \left(\begin{array}{l} a \to P \wedge \{a\} \subseteq \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ [Comm:def]:p. 34 ”} \\
 & CSP2 \left(\begin{array}{l} CSP1 \left(ok' \wedge \mathbf{R3} \left(WTC(a) \lhd wait' \rhd \left(\begin{array}{l} state' = state \wedge \\ WTC(a); EVTSNOW\{a\} \end{array} \right) \right) \right); P \\ \wedge \{a\} \subseteq \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ Restrictions on refusals. See [Lemma7]:p. 134 for details ”} \\
 & CSP2 \left(\begin{array}{l} CSP1 \left(ok' \wedge \mathbf{R3} \left(\begin{array}{l} \neg wait' \wedge state' = state \wedge \\ WTC(a); EVTSNOW\{a\} \end{array} \right) \right); P \\ \wedge \{a\} \subseteq \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ [Hid:def]:p. 39 ”} \\
 & CSP2 \left(\begin{array}{l} CSP1 \left(ok' \wedge \mathbf{R3} \left(\begin{array}{l} \neg wait' \wedge state' = state \wedge \\ WTC(a); EVTSNOW\{a\} \end{array} \right) \right); P \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ Prefixing is } \mathbf{R3}\text{, Q is } \mathbf{R3}\text{, Choice preserves } \mathbf{R3} \text{ ”} \\
 & CSP2 \circ \mathbf{R3} \left(\begin{array}{l} CSP1 \left(ok' \wedge \neg wait' \wedge state' = state \wedge WTC(a); EVTSNOW\{a\} \right); P \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ [CSP1:def]:p. 26 case split, Q and P are CSP1 ”}
\end{array}
$$

$\equiv$ " CSP1 case split, Q and P are CSP1 "

$$CSP2 \circ \mathbf{R}3 \left(\begin{array}{l} \neg ok \wedge GROW \wedge \\ \left(\begin{array}{l} (\neg ok \wedge GROW \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\vee$

$$CSP2 \circ \mathbf{R}3 \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge \neg wait' \wedge state' = state \wedge WTC(a); EVTSNOW\{a\} \end{array} \right); P \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " [NEV:def]:p. 29 "

$$CSP2 \circ \mathbf{R}3 \left(\begin{array}{l} \neg ok \wedge GROW \wedge \\ \left(\begin{array}{l} \neg ok \wedge GROW \vee \\ \left(\begin{array}{l} (\neg ok \wedge NOEVTS(slots, slots')); \\ \left(\begin{array}{l} slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\vee$

$$CSP2 \circ \mathbf{R}3 \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge \neg wait' \wedge state' = state \wedge WTC(a); EVTSNOW\{a\} \end{array} \right); P \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " Logic "

$$CSP2 \circ \mathbf{R}3 \left(\begin{array}{l} \neg ok \wedge GROW \end{array} \right) \setminus \{a\}$$

$\vee$

$$CSP2 \circ \mathbf{R}3 \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge \neg wait' \wedge state' = state \wedge WTC(a); EVTSNOW\{a\} \end{array} \right); P \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " [CSP1:def]:p. 26 "

$$CSP2 \circ \mathbf{R}3 \circ CSP1 \left(\begin{array}{l} \left(\begin{array}{l} ok' \wedge \neg wait' \wedge state' = state \wedge WTC(a); EVTSNOW\{a\} \end{array} \right); P \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \left(\begin{array}{l} IMMEVTS \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " an event a has to be performed, Lemma 2 "

$\equiv$ " an event a has to be performed, Lemma 2 "

$$CSP2 \circ \mathbf{R}3 \circ CSP1 \left(\begin{array}{l} \Big(ok' \wedge \neg wait' \wedge state' = state \wedge WTC(a); EVTSNOW\{a\} \Big); P \wedge \\ \left(\begin{array}{l} (Q \wedge NOEVTS) \\ ; \\ \Big(IMMEVTS \Big) \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " Event is performed in the first slot "

$$CSP1 \circ CSP2 \left(\begin{array}{l} (ok \wedge ok' \wedge \neg wait' \wedge state' = state \wedge EVTSNOW\{a\}(slots, slots')); P \wedge \\ \left(\begin{array}{l} (ok \wedge Q \wedge \sharp slots = \sharp slots' \wedge NOEVTS(slots, slots')) \\ ; IMMEVTS(slots, slots') \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " Lemma 6 "

$$CSP1 \circ CSP2 \left(\begin{array}{l} (ok \wedge ok' \wedge \neg wait' \wedge state' = state \wedge EVTSNOW\{a\}(slots, slots')); P \wedge \\ \Big((ok \wedge Q \wedge slots \cong slots'); IMMEVTS(slots, slots') \Big) \end{array} \right) \setminus \{a\}$$

$\equiv$ " R3 from hiding def - Lemma 4 "

$$CSP1 \circ CSP2 \left(\begin{array}{l} (ok \wedge ok' \wedge \neg wait' \wedge state' = state \wedge EVTSNOW\{a\}(slots, slots')); P \wedge \\ \Big((ok \wedge \neg wait \wedge Q \wedge slots \cong slots'); IMMEVTS(slots, slots') \Big) \end{array} \right) \setminus \{a\}$$

$\equiv$ " Seq comp def "

$$CSP1 \circ CSP2 \left(\begin{array}{l} (ok \wedge ok' \wedge \neg wait' \wedge state' = state \wedge EVTSNOW\{a\}(slots, slots')); P \wedge \\ \left(\begin{array}{l} \exists obs_0 \bullet ok \wedge \neg wait \wedge Q[obs_0/obs'] \wedge slots \cong slots_0 \wedge \\ IMMEVTS(slots_0, slots') \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " Lemma 7 "

$$CSP1 \circ CSP2 \left(\begin{array}{l} (ok \wedge ok' \wedge \neg wait' \wedge state' = state \wedge EVTSNOW\{a\}(slots, slots')); P \wedge \\ \left(\begin{array}{l} \exists obs_0 \bullet ok \wedge \neg wait \wedge Q[obs_0/obs'] \wedge slots \cong slots_0 \wedge \\ IMMEVTS(slots, slots') \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " Logic "

$$CSP1 \circ CSP2 \left(\begin{array}{l} (ok \wedge ok' \wedge \neg wait' \wedge state' = state \wedge EVTSNOW\{a\}(slots, slots')); P \wedge \\ \Big(\exists obs_0 \bullet ok \wedge \neg wait \wedge Q[obs_0/obs'] \wedge slots \cong slots_0 \Big) \wedge \\ IMMEVTS(slots, slots') \end{array} \right) \setminus \{a\}$$

$\equiv$ " Assuption "

$$CSP1 \circ CSP2 \left(\begin{array}{l} (ok \wedge ok' \wedge \neg wait' \wedge state' = state \wedge EVTSNOW\{a\}(slots, slots')); P \wedge \\ IMMEVTS(slots, slots') \end{array} \right) \setminus \{a\}$$

$\equiv$ " Lemma 8, Lemma 9 "

$$CSP1 \circ CSP2 \Big((ok \wedge ok' \wedge \neg wait' \wedge state' = state \wedge EVTSNOW\{a\}(slots, slots')); P \Big) \setminus \{a\}$$

$\equiv$ " Prefix def "

$\equiv$ " Prefix def "

$CSP1 \circ CSP2 \Big(\; a \rightarrow Skip; P \wedge \{a\} \subset \bigcap srefs(slots' \lessdot\!\!\lessdot slots) \; \Big) \setminus \{a\}$

$\equiv$ " Hiding def, P and prefix are healthy "

$(a \rightarrow Skip; P) \setminus \{a\}$

$\equiv$ " Law oh hiding "

$a \rightarrow Skip \setminus \{a\}; P \setminus \{a\}$

$\equiv$ " Law oh hiding "

$Skip; P \setminus \{a\}$

$\equiv$ " Skip is a left unit for ; "

$P \setminus \{a\}$

Case 3

High priority process is ready to perform an events a so refusals of the low priority one are are "marked" with a and he is not refusing to do it. Hiding constrains the refusals and in order to make a internal it is not allowing to wait for it (a has to be refused). For that reason the whole case is falsified and turns into reactive miracle.

[WeakChoice:reduce] $WeakChoice(Q, a \to P) \setminus \{a\} \equiv \mathbf{R3}(CSP1(False))$

$$
\begin{array}{ll}
 & WeakChoice(Q, a \to P) \setminus \{a\} \\
\equiv & \text{" [pExt:def:WeakChoice]:p. 56 "} \\
 & CSP2(Q \wedge \\
 & \left(\begin{array}{l} \left(\; a \to P \wedge NOEVTS(slots, slots') \wedge wait' \; \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{" [Hid:def]:p. 39, [CSP2:def]:p. 27 "} \\
 & CSP2(Q \wedge \{a\} \subseteq \bigcap Refs(slots' \setminus\!\!\setminus slots) \\
 & \left(\begin{array}{l} \left(\; a \to P \wedge NOEVTS(slots, slots') \wedge wait' \; \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{" Set theory, Lemma3 "} \\
 & CSP2(Q \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \\
 & \left(\begin{array}{l} \left(\begin{array}{l} a \to P \wedge NOEVTS(slots, slots') \\ \wedge wait' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{" [Comm:def]:p. 34, P is R3 "} \\
 & CSP2(Q \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \\
 & \left(\begin{array}{l} \left(\begin{array}{l} CSP1(ok' \wedge \mathbf{R3}(WTC(a))) \wedge NOEVTS(slots, slots') \\ \wedge wait' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{" Properties of CSP1 and R3 "}
\end{array}
$$

$\equiv$ " Properties of CSP1 and R3 "

$$CSP2(Q \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \left(\begin{array}{l} \left(\begin{array}{l} CSP1(ok' \wedge \mathbf{R3}(WTC(a) \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots))) \wedge \\ NOEVTS(slots, slots') \wedge \\ wait' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " [WTC:def]:p. 34 "

$$CSP2(Q \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \left(\begin{array}{l} \left(\begin{array}{l} CSP1(ok' \wedge \mathbf{R3}(False)) \wedge \\ NOEVTS(slots, slots') \wedge \\ wait' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " [CSP1:def]:p. 26, [R3:def]:p. 26 "

$$CSP2(Q \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \left(\begin{array}{l} \left(\begin{array}{l} (\neg ok \wedge slots \preccurlyeq slots' \vee ok \wedge ok' \wedge wait' \wedge wait \wedge state' = state \wedge slots' = slots) \wedge \\ NOEVTS(slots, slots') \wedge wait' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " R3 in hiding def "

$$CSP2(Q \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \neg wait \wedge \left(\begin{array}{l} \left(\begin{array}{l} \neg ok \wedge slots \preccurlyeq slots' \wedge \\ NOEVTS(slots, slots') \wedge wait' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " [Seq:def]:p. 33 "

$$CSP2(\neg ok \wedge Q \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \neg wait \wedge \left(\begin{array}{l} \left(\begin{array}{l} slots \preccurlyeq slots' \wedge \\ NOEVTS(slots, slots') \wedge wait' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) \setminus \{a\}$$

$\equiv$ " Q is CSP1 "

$\equiv$ " Q is CSP1 "

$$CSP2(\neg ok \wedge slots \preccurlyeq slots' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \neg wait \wedge \left(\begin{array}{l} \left(\begin{array}{l} \wedge slots \preccurlyeq slots' \wedge \\ NOEVTS(slots, slots') \wedge wait' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))\setminus\{a\}$$

$\equiv$ " Lemma 3 "

$$CSP2(\neg ok \wedge slots \preccurlyeq slots' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \neg wait \wedge \left(\begin{array}{l} \left(\begin{array}{l} slots \preccurlyeq slots' \wedge \\ NOEVTS(slots, slots') \wedge wait' \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))\setminus\{a\}$$

$\equiv$ " NOEVTS is GROW "

$$CSP2(\neg ok \wedge slots \preccurlyeq slots' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \neg wait \wedge \left(\begin{array}{l} NOEVTS(slots, slots') \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))\setminus\{a\}$$

$\equiv$ " refusals ar unconstrained so PRI is irrelevant "

$$CSP2(\neg ok \wedge slots \preccurlyeq slots' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \neg wait \wedge \left(\begin{array}{l} NOEVTS(slots, slots') \\ ; \\ \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))\setminus\{a\}$$

$\equiv$ " refusals ar unconstrained so $E \subseteq eqvrefs(slots)$ is irrelevant "

$$CSP2(\neg ok \wedge slots \preccurlyeq slots' \wedge a \in \bigcap srefs(slots' \setminus\!\!\setminus slots) \wedge \neg wait \wedge \left(\begin{array}{l} NOEVTS(slots, slots') \\ ; \\ \left(\begin{array}{l} \exists E \bullet \left(FSTEVTS(E)(slots, slots') \wedge E \neq \emptyset \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))\setminus\{a\}$$

$\equiv$ " [IME:def]:p. 30 "

$$CSP2(\neg ok \wedge slots \preccurlyeq slots' \wedge \left(\begin{array}{l} NOEVTS(slots, slots') \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots = slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right))\setminus\{a\}$$

$\equiv$ " [CSP2:def]:p. 27 "

$$
\begin{array}{ll}
\equiv & \text{“ [CSP2:def]:p. 27 ”} \\
& \exists ok_0 (\neg ok \wedge slots \preccurlyeq slots' \wedge \\
& \left(\begin{array}{l} NOEVTS(slots, slots') \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots = slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ Logic ”} \\
& (\neg ok \wedge slots \preccurlyeq slots' \wedge \\
& \left(\begin{array}{l} NOEVTS(slots, slots') \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots = slots' \end{array} \right) \end{array} \right) \setminus \{a\} \\
\equiv & \text{“ Lemma 5 ”} \\
& (\neg ok \wedge slots \preccurlyeq slots') \setminus \{a\} \\
\equiv & \text{“ [CSP1:def]:p. 26 ”} \\
& CSP1(False) \setminus \{a\} \\
\equiv & \text{“ [Hid:def]:p. 39 ”} \\
& \mathbf{R}3(CSP1(False))
\end{array}
$$

B.4.3 Lemmas

Lemma 1

If P;Q performed no events then P performed no events as well.

$$(P;Q) \wedge NOEVTS(slots, slots') \equiv ((P \wedge NOEVTS(slots, slots')); Q) \wedge NOEVTS(slots, slots')$$

Lemma 2

You can not perform an event and not perform it at the same time.

$$EVTSNOW\{E\}(slots, slots') \wedge NOEVTS(slots, slots') \equiv False$$

Lemma 3

This lemma is important for the prioritised theory. In other words it is showing how constrains on refusals propagate via sequential composition $(P;Q)$ when Q is *PRI* healthy. It is used whenever we want to use hiding on sequentially composed processes.

[ref:mark:law] $\forall slots_1, slots_2, ref \bullet$

$$
\begin{array}{l}
\left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ head(srefs(slots_2 \setminus\!\setminus slots_1)) \subseteq last(srefs(slots_1)) \end{array} \right) \\
\Rightarrow \\
\left(ref \subseteq \bigcap srefs(slots_2) \Leftrightarrow ref \subseteq \bigcap srefs(slots_1) \wedge ref \subseteq \bigcap srefs(slots_2 \setminus\!\setminus slots_1) \right)
\end{array}
$$

$$
\begin{array}{l}
\forall slots_1, slots_2, hidn \bullet \\
\left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ head(srefs(slots_2 \nwarrow slots_1)) \subseteq last(srefs(slots_1)) \end{array} \right) \\
\Rightarrow \\
\left(hidn \subseteq \bigcap srefs(slots_2) \Leftrightarrow hidn \subseteq \bigcap srefs(slots_1) \wedge hidn \subseteq \bigcap srefs(slots_2 \nwarrow slots_1) \right)
\end{array}
$$

$\equiv$ “ [EX:pfx]:p. 108 ”

$$
\begin{array}{l}
\forall slots_1, slots_2, hidn \exists pfx, sfx, slot, slot' \bullet \\
\left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ head(srefs(slots_2 \nwarrow slots_1)) \subseteq last(srefs(slots_1)) \\ slots_1 = pfx \frown \langle slot \rangle \wedge slots_2 = pfx \frown \langle slot' \rangle \frown sfx \end{array} \right) \\
\Rightarrow \\
\left(hidn \subseteq \bigcap srefs(slots_2) \Leftrightarrow hidn \subseteq \bigcap srefs(slots_1) \wedge hidn \subseteq \bigcap srefs(slots_2 \nwarrow slots_1) \right)
\end{array}
$$

$\equiv$ “ [DF:pfx]:p. 110 ”

$$
\begin{array}{l}
\forall slots_1, slots_2, hidn \exists pfx, sfx, slot, slot' \bullet \\
\left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ head(srefs(slots_2 \nwarrow slots_1)) \subseteq last(srefs(slots_1)) \\ slots_1 = pfx \frown \langle slot \rangle \wedge slots_2 = pfx \frown \langle slot' \rangle \frown sfx \end{array} \right) \\
\Rightarrow \\
\left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx \frown \langle slot' \rangle \frown sfx) \\ \Leftrightarrow \\ hidn \subseteq \bigcap srefs(pfx \frown \langle slot \rangle) \wedge hidn \subseteq \bigcap srefs(ssub(slot', slot) : sfx) \end{array} \right)
\end{array}
$$

$\equiv$ “ Set theory ”

$$
\begin{array}{l}
\forall slots_1, slots_2, hidn \exists pfx, sfx, slot, slot' \bullet \\
\left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ head(srefs(ssub(slot', slot) : sfx)) \subseteq last(srefs(pfx \frown \langle slot \rangle)) \\ slots_1 = pfx \frown \langle slot \rangle \wedge slots_2 = pfx \frown \langle slot' \rangle \frown sfx \end{array} \right) \\
\Rightarrow \\
\left(\begin{array}{l} \left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx) \wedge \\ hidn \subseteq sref(slot') \wedge \\ hidn \subseteq \bigcap srefs(sfx) \end{array} \right) \\ \Leftrightarrow \\ \left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx) \wedge \\ hidn \subseteq sref(slot) \wedge \\ hidn \subseteq sref(ssub(slot', slot)) \wedge \\ hidn \subseteq \bigcap srefs(sfx) \end{array} \right) \end{array} \right)
\end{array}
$$

$\equiv$ “ [RFS:def]:p. 107 , properties of map function ”

$$
\begin{array}{ll}
 & \forall slots_1, slots_2, hidn \exists pfx, sfx, slot, slot' \bullet \\
 & \left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ sref(ssub(slot', slot)) \subseteq sref(slot) \\ slots_1 = pfx \frown \langle slot \rangle \wedge slots_2 = pfx \frown \langle slot' \rangle \frown sfx \end{array} \right) \\
 & \Rightarrow \\
 & \left(\begin{array}{l} \left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx) \wedge \\ hidn \subseteq sref(slot') \wedge \\ hidn \subseteq \bigcap srefs(sfx) \end{array} \right) \\ \Leftrightarrow \\ \left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx) \wedge \\ hidn \subseteq sref(slot) \wedge \\ hidn \subseteq sref(ssub(slot', slot)) \wedge \\ hidn \subseteq \bigcap srefs(sfx) \end{array} \right) \end{array} \right) \\
\equiv & \quad \text{“ [ssub:ref]:p. } \mathbf{??} \text{ ”} \\
 & \forall slots_1, slots_2, hidn \exists pfx, sfx, slot, slot' \bullet \\
 & \left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ sref(slot') \subseteq sref(slot) \\ slots_1 = pfx \frown \langle slot \rangle \wedge slots_2 = pfx \frown \langle slot' \rangle \frown sfx \end{array} \right) \\
 & \Rightarrow \\
 & \left(\begin{array}{l} \left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx) \wedge \\ hidn \subseteq sref(slot') \wedge \\ hidn \subseteq \bigcap srefs(sfx) \end{array} \right) \\ \Leftrightarrow \\ \left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx) \wedge \\ hidn \subseteq sref(slot) \wedge \\ hidn \subseteq sref(slot') \wedge \\ hidn \subseteq \bigcap srefs(sfx) \end{array} \right) \end{array} \right) \\
\equiv & \quad \text{“ } \subseteq \text{ is transitive ”} \\
 & \forall slots_1, slots_2, hidn \exists pfx, sfx, slot, slot' \bullet \\
 & \left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ sref(slot') \subseteq sref(slot) \\ slots_1 = pfx \frown \langle slot \rangle \wedge slots_2 = pfx \frown \langle slot' \rangle \frown sfx \end{array} \right) \\
 & \Rightarrow \\
 & \left(\begin{array}{l} \left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx) \wedge \\ hidn \subseteq sref(slot') \wedge \\ hidn \subseteq \bigcap srefs(sfx) \\ hidn \subseteq sref(slot) \wedge \end{array} \right) \\ \Leftrightarrow \\ \left(\begin{array}{l} hidn \subseteq \bigcap srefs(pfx) \wedge \\ hidn \subseteq sref(slot) \wedge \\ hidn \subseteq sref(slot') \wedge \\ hidn \subseteq \bigcap srefs(sfx) \end{array} \right) \end{array} \right) \\
\equiv & \quad \text{“ Logic ”}
\end{array}
$$

$$
\begin{array}{ll}
 & \forall slots_1, slots_2, hidn \exists pfx, sfx, slot, slot' \bullet \\
 & \left(\begin{array}{l} slots_1 \preccurlyeq slots_2 \wedge \\ sref(slot') \subseteq sref(slot) \\ slots_1 = pfx \frown \langle slot \rangle \wedge slots_2 = pfx \frown \langle slot' \rangle \frown sfx \end{array} \right) \\
 & \Rightarrow \\
 & \left(\begin{array}{l} True \end{array} \right) \\
\equiv & \quad \text{“ Logic ”} \\
 & True
\end{array}
$$

Lemma 4

$$\mathbf{R}3(P) \equiv \mathbf{R}3(\neg wait \wedge P)$$

Lemma 5

A useful property of any **R**1 healthy process.

$$GROW$$

$$\Downarrow$$

$$(NOEVTS(slots, slots'); (IMMEVTS(slots, slots') \vee slots \cong slots'))$$

Lemma 6

$$NOEVTS(slots, slots') \wedge \#slots = \#slots' \equiv slots \cong slots'$$

Lemma 7

$$s1 \cong s2 \wedge IMMEVTS(s2, s3) \equiv s1 \cong s2 \wedge IMMEVTS(s1, s3)$$

Lemma 8 ?

$$(EVTSNOW\{E\}(slots, slots') \wedge E \neq \emptyset) \Rightarrow IMMEVTS(slots, slots')$$

Lemma 9

$$IMMEVTS(slots, slots'); GROW \equiv IMMEVTS(slots, slots')$$

Lemma 10

Important for proving $(P;Q)\diagdown hidn \equiv P\diagdown hidn; Q\diagdown hidn$. It shows the main difference between treating refusals in S-C and prioritized S-C. It gives us an idea of how **R**2 interacts with *CSP*3 and what do we need *CSP*3 for. It is not used explicitly in any of the proves above.

$$
\begin{array}{ll}
& P \wedge \neg wait \wedge ok \\
\equiv & \\
& \left(\begin{array}{l} \exists ss \bullet head(srefs(ss \nwarrow slots)) \subseteq last(srefs(slots)) \wedge \\ P[ss, ss \,\#\!\#\, (slots' \nwarrow slots)/slots, slots'] \end{array} \right) \wedge \neg wait \wedge ok
\end{array}
$$

$$
\begin{array}{ll}
& P \wedge \neg wait \wedge ok \\
\equiv & \text{“ P is CSP3 ”} \\
& Skip; P \wedge \neg wait \wedge ok \\
\equiv & \text{“ [Skip:def]:p. 31 ”} \\
& (PRI(state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots'); P) \wedge \neg wait \wedge ok \\
\equiv & \text{“ [PRI:def]:p. 57 ”} \\
& \left(\begin{array}{l} (ok \Rightarrow head(srefs(slots' \nwarrow slots)) \subseteq eqvref(slots)) \wedge \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots' \end{array} \right); P) \wedge \neg wait \wedge ok \\
\equiv & \text{“ [Seq:def]:p. 33, [eqvref:def]:p. \textbf{??} ”} \\
& \left(\begin{array}{l} head(srefs(slots' \nwarrow slots)) \subseteq last(srefs(slots)) \wedge \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots' \end{array} \right); P) \wedge \neg wait \wedge ok \\
\equiv & \text{“ P is R2 ”} \\
& \left(\begin{array}{l} \left(\begin{array}{l} head(srefs(slots' \nwarrow slots)) \subseteq last(srefs(slots)) \wedge \\ state = state' \wedge \neg wait' \wedge ok' \wedge slots \cong slots' \end{array} \right) \\ ; \\ \mathbf{R}2(P) \end{array} \right) \wedge \neg wait \wedge ok \\
\equiv & \text{“ [Seq:def]:p. 33 ”} \\
& \left(\begin{array}{l} \exists obs_0 \bullet head(srefs(slots_0 \nwarrow slots)) \subseteq last(srefs(slots)) \wedge \\ state = state_0 \wedge \neg wait_0 \wedge ok_0 \wedge slots \cong slots_0 \wedge \\ \mathbf{R}2(P)[obs_0/obs] \end{array} \right) \wedge \neg wait \wedge ok \\
\equiv & \text{“ One point rule ”} \\
& \left(\begin{array}{l} \exists slots_0 \bullet head(srefs(slots_0 \nwarrow slots)) \subseteq last(srefs(slots)) \wedge \\ slots \cong slots_0 \wedge \\ \mathbf{R}2(P)[slots_0/slots] \end{array} \right) \wedge \neg wait \wedge ok \\
\equiv & \text{“ [R2:def]:p. 26 ”} \\
& \left(\begin{array}{l} \exists slots_0, ss \bullet head(srefs(slots_0 \nwarrow slots)) \subseteq last(srefs(slots)) \wedge \\ slots \cong slots_0 \wedge eqvref(ss) = eqvref(slots_0) \wedge \\ P[ss, ss \,\#\!\#\, (slots' \nwarrow slots_0)/slots, slots'] \end{array} \right) \wedge \neg wait \wedge ok \\
\equiv & \text{“ Property of eqvref ”} \\
& \left(\begin{array}{l} \exists slots_0, ss \bullet head(srefs(ss \nwarrow slots)) \subseteq last(srefs(slots)) \wedge \\ slots \cong slots_0 \wedge eqvref(ss) = eqvsef(slots_0) \wedge \\ P[ss, ss \,\#\!\#\, (slots' \nwarrow slots_0)/slots, slots'] \end{array} \right) \wedge \neg wait \wedge ok \\
\equiv & \text{“ Property of slots substraction and } \cong \text{ ”}
\end{array}
$$

$$\left(\begin{array}{l} \exists slots_0, ss \bullet head(srefs(ss \lessdot slots)) \subseteq last(srefs(slots)) \wedge \\ slots \cong slots_0 \wedge eqvref(ss) = eqvref(slots_0) \wedge \\ P[ss, ss \# (slots' \lessdot slots)/slots, slots'] \end{array}\right) \wedge \neg wait \wedge ok$$

$\equiv$ " There always exists $slots_0$ satisfying above conditions "

$$\left(\begin{array}{l} \exists ss \bullet head(srefs(ss \lessdot slots)) \subseteq last(srefs(slots)) \wedge \\ P[ss, ss \# (slots' \lessdot slots)/slots, slots'] \end{array}\right) \wedge \neg wait \wedge ok$$

Lemma 11

$$[P \wedge Q] \Rightarrow [P]$$

$$\begin{array}{ll} & [P \wedge Q] \Rightarrow [P] \\ \equiv & \\ & \neg [P \wedge Q] \vee [P] \\ \equiv & \\ & \exists obs \bullet (\neg P \vee \neg Q) \vee \\ & \forall obs \bullet P \\ \equiv & \\ & \exists obs \bullet \neg P \vee \\ & \exists obs \bullet \neg Q \vee \\ & \forall obs \bullet P \\ \equiv & \\ & \exists obs \bullet \neg P \vee \\ & \neg \exists obs \bullet \neg P \vee \\ & \exists obs \bullet \neg Q \vee \\ \equiv & \\ & True \end{array}$$

B.4.4 Prioritised Choice Implements External Choice

$$[\text{pECimplementsEC}] \qquad [(P \overleftarrow{\Box} Q) \Rightarrow (P \Box Q)]$$

$$
\begin{array}{ll}
 & [(P \overleftarrow{\Box} Q) \Rightarrow (P \Box Q)] \\
\equiv & \text{“ [Ext:def]:p. 35 ”} \\
 & [(P \overleftarrow{\Box} Q) \Rightarrow \\
 & (P \wedge Q \wedge Stop \vee Choice(P,Q) \vee Choice(Q,P)) \\
\equiv & \text{“ [pExt:def]:p. 56 ”} \\
 & [(P \wedge Q \wedge Stop \vee Choice(P,Q) \vee WeakChoice(Q,P)) \\
 & \Rightarrow \\
 & (P \wedge Q \wedge Stop \vee Choice(P,Q) \vee Choice(Q,P))] \\
\equiv & \text{“ Logic ”} \\
 & WeakChoice(Q,P) \\
 & \Rightarrow \\
 & (P \wedge Q \wedge Stop \vee Choice(P,Q) \vee Choice(Q,P)) \\
\Uparrow & \\
 & WeakChoice(Q,P) \\
 & \Rightarrow \\
 & Choice(Q,P) \\
\equiv & \text{“ [Choice:def]:p. 36 ”} \\
 & WeakChoice(Q,P) \\
 & \Rightarrow \\
 & CSP2(Q \wedge \left(\begin{array}{l} (P \wedge NOEVTS(slots, slots')) \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right)) \\
\equiv & \text{“ [pExt:def:WeakChoice]:p. 56 ”} \\
 & CSP2(Q \wedge \\
 & \left(\begin{array}{l} \left(\begin{array}{l} P \wedge NOEVTS(slots, slots') \wedge wait' \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right)) \\
 & \Rightarrow \\
 & CSP2(Q \wedge \left(\begin{array}{l} (P \wedge NOEVTS(slots, slots')) \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right)) \\
\equiv & \text{“ CSP2:def ”}
\end{array}
$$

$$
\begin{array}{ll}
 & (\exists ok_0 \bullet (Q \wedge \\
 & \left(\begin{array}{l} \left(\begin{array}{l} P \wedge NOEVTS(slots, slots') \wedge wait' \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) [ok_0/ok'] \wedge ok_0 \Rightarrow ok') \\
 & \Rightarrow \\
 & (\exists ok_0 \bullet (Q \wedge \left(\begin{array}{l} (P \wedge NOEVTS(slots, slots')) \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) [ok_0/ok'] \wedge ok_0 \Rightarrow ok') \\
\equiv & \quad \text{“ Logic ”} \\
 & \exists ok_0 \bullet (\\
 & ((Q \wedge \\
 & \left(\begin{array}{l} \left(\begin{array}{l} P \wedge NOEVTS(slots, slots') \wedge wait' \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) [ok_0/ok'] \wedge ok_0 \Rightarrow ok') \\
 & \Rightarrow \\
 & ((Q \wedge \left(\begin{array}{l} (P \wedge NOEVTS(slots, slots')) \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) [ok_0/ok'] \wedge ok_0 \Rightarrow ok')) \\
\equiv & \quad \text{“ Logic ”} \\
 & \exists ok_0 \bullet (\\
 & \left(\begin{array}{l} \left(\begin{array}{l} P \wedge NOEVTS(slots, slots') \wedge wait' \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) [ok_0/ok'] \\
 & \Rightarrow \\
 & \left(\begin{array}{l} (P \wedge NOEVTS(slots, slots')) \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok') \end{array} \right) \end{array} \right) [ok_0/ok']) \\
\equiv & \quad \text{“ Logic ”}
\end{array}
$$

$$
\begin{array}{ll}
 & \exists ok_0 \bullet (\\
 & \left(\begin{array}{l} \left(\begin{array}{l} P \wedge NOEVTS(slots, slots') \wedge wait' \end{array} \right) \\ ; \\ PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right) \end{array} \right) \\
\Rightarrow & \\
 & \left(\begin{array}{l} (P \wedge NOEVTS(slots, slots') \wedge wait') \\ ; \\ \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right) \end{array} \right)) \\
\Uparrow & \\
 & \exists ok_0 \bullet (\\
 & PRI \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right) \\
 & \Rightarrow \\
 & \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right)) \\
\Uparrow & \\
 & \exists ok_0 \bullet (\\
 & \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right) \\
 & \Rightarrow \\
 & \left(\begin{array}{l} IMMEVTS(slots, slots') \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right)) \\
\equiv & \quad \text{“ [IME:def]:p. 30 ”} \\
 & \exists ok_0 \bullet (\\
 & \left(\begin{array}{l} \exists E \bullet \left(\begin{array}{l} FSTEVTS(E)(slots, slots') \\ \wedge E \neq \emptyset \wedge E \subseteq eqvrefs(slots) \end{array} \right) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right) \\
 & \Rightarrow \\
 & \left(\begin{array}{l} \exists E \bullet (FSTEVTS(E)(slots, slots') \wedge E \neq \emptyset) \\ \vee slots \cong slots' \wedge (\neg wait' \vee \neg ok_0) \end{array} \right)) \\
\equiv & \quad \text{“ Logic ”} \\
 & True
\end{array}
$$

Bibliography

[Aky06] Ian F. Akyildiz. Grand challenges for wireless sensor networks. In Reinhard German and Armin Heindl, editors, *MMB*, pages 13–14. VDE Verlag; VDE Verlag, 2006.

[Ana93] Paul Anand. The philosophy of intransitive preference. *Economic Journal*, 103(417):337–46, March 1993.

[Bar89] G Barrett. The semantics of priority and fairness in OCCAM. In *Proceedings of Mathematical Foundations Of Programming Semantics, Fifth International Conference, New Orleans,USA*, pages 194–209, March/April 1989.

[BCG86] Gérard Berry, P. Couronné, and Georges Gonthier. Synchronous programming of reactive systems: An introduction to ESTEREL. Technical Report 647, INRIA, 1986.

[BG09] Andrew Butterfield and Paweł Gancarski. slotted-Circus: A generic UTP framework for discretely-timed Circus. Technical Report TCD-CS-09-32, School of Computer Science & Statistic Trinity College Dublin, Trinity College, Dublin 2, Ireland, July 2009.

[BGJK88a] Geoff Barrett, Michael Goldsmith, Geraint Jones, and A. Kay. The meaning and implementation of PRI ALT in occam. In *OUG-9: Occam and the Transputer – Research and Applications*, pages 37–46, September 1988.

[BGJK88b] Geoff Barrett, Michael Goldsmith, Geraint Jones, and Andrew Kay. The meaning and implementation of PRI ALT in occam. In *occam and the Transputer, research and applications (OUG-9)*, 1988.

[BGW09] Andrew Butterfield, Paweł Gancarski, and Jim Woodcock. State visibility and communication in Unifying Theories of Programming. In *TASE*, pages 47–54. IEEE Computer Society, 2009.

[BHM88] Maya Bar-Hillel and Avishai Margalit. How vicious are cycles of intransitive choice? *Theory and Decision*, 24:119–145, 1988. 10.1007/BF00132458.

[BSW07] Andrew Butterfield, Adnan Sherif, and Jim Woodcock. slotted-Circus. In *IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 75–97. Springer, 2007.

[But06] Andrew Butterfield. Issues regarding R2 and undefinedness. http://www.scm.tees.ac.uk/utpsymposium/slides/abutterfield_utp06_slides.pdf, 2006.

[But10] Andrew Butterfield. Saoithín: A theorem prover for UTP. In *UTP*, volume 6445. Springer, 2010.

[But11a] Andrew Butterfield. A denotational semantics for handel-C. *Formal Asp. Comput*, 23(2):153–170, 2011.

[But11b] Andrew Butterfield. Formal methods, lecture slides. http://www.scss.tcd.ie/Andrew.Butterfield/Teaching/CS4003/CS4003-Week-3-4up.pdf, 2011.

[BW02] A Butterfield and J Woodcock. Semantics of prialt in handel-C. In *Concurrent Systems Engineering, Proceedings of the Conference on Communicating Processing Architectures*. IOS Press, 2002.

[BW05a] Andrew Butterfield and Jim Woodcock. prialt in handel-C: an operational semantics. *STTT*, 7(3):248–267, 2005.

[BW05b] Andrew Butterfield and Jim Woodcock. prialt in handel-C: an operational semantics. *STTT*, 7(3):248–267, 2005.

[BW06a] Andrew Butterfield and Jim Woodcock. A ”hardware compiler” semantics for handel-C. *Electr. Notes Theor. Comput. Sci*, 161:73–90, 2006.

[BW06b] Andrew Butterfield and Jim Woodcock. A ”hardware compiler” semantics for handel-C. *Electr. Notes Theor. Comput. Sci*, 161:73–90, 2006.

[Cel05] Celoxica. *Handel-C Language Reference Manual*. Celoxica, 2005.

[CGJ+01] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000:176, 2001.

[CH88] R. Cleaveland and M. Hennessy. Priorities in process algebra. In *Proceedings* 3^{th} *Annual Symposium on Logic in Computer Science,* Edinburgh, pages 193–202. IEEE Computer Society Press, 1988.

[Chu10] Verma Churiwala, Kumar. Combinational loops: Exploring the types of combinational loops. http://www.eetindia.co.in/STATIC/PDF/201003/EEIOL_2010MAR18_SIG_TA_01.pdf, 2010.

[CLN] R. Cleaveland, G. Luettgen, and V. Natarajan. *Priority in Process Algebra*.

[CNSL96] Rance Cleaveland, V. Natarajan, Steve Sims, and Gerald Lüttgen. Modeling and verifying distributed systems using priorities: A case study. *Software - Concepts and Tools*, 17(2):50–62, 1996.

[Con85] J.A.N.C. Condorcet. *Essai sur l’application de l’analyse à la probabilité des décisions rendues à la pluralité des voix*. Chelsea Publishing Series. Chelsea Pub. Co., 1785.

[CW95] Camilleri and Winskel. CCS with priority choice. *INFCTRL: Information and Computation (formerly Information and Control)*, 116, 1995.

[DGM09] David Déharbe, Stephenson Galvão, and Anamaria Martins Moreira. Formalizing freeRTOS: First steps. In *SBMF*, volume 5902. Springer, 2009.

[Fid93] C. J. Fidge. A formal definition of priority in CSP. *ACM Transactions on Programming Languages and Systems*, 15(4):681–705, September 1993.

[GB09] Paweł Gancarski and Andrew Butterfield. The denotational semantics of slotted-circus. In *FM*, volume 5850. Springer, 2009.

[GB10] Paweł Gancarski and Andrew Butterfield. Prioritized slotted-Circus. In *ICTAC*, volume 6255. Springer, 2010.

[GR05] Ajeesh Gopalakrishnan and Bart Rem. Automatic handel-C generation from MATLAB(tm) and simulink(tm) for motion control with an FPGA. In *Communicating Process Architectures 2005: WoTUG-28*. IOS Press, Amsterdam, 2005.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HH98] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Series in Computer Science. Prentice Hall, 1998.

[HLMS07] Tony Hoare, Gary T. Leavens, Jayadev Misra, and Natarajan Shankar. The verified software initiative: A manifesto. http://qpq.csl.sri.com/vsr/manifesto.pdf, 2007.

[HM05] Hoare and Milner. Grand challenges for computing research. *COMPJ: The Computer Journal*, 48, 2005.

[Hoa78] Tony Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85a] C. A. R. Hoare. *Communicating Sequential Processes*. Intl. Series in Computer Science. Prentice Hall, 1985.

[Hoa85b] C. A. R. Hoare. Programs are predicates. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 141–155. Prentice-Hall, Inc., 1985.

[INM88] INMOS Limited. *occam 2 Reference Manual*, 1988.

[IS07] Wilson Ifill and Steve Schneider. A step towards refining and translating B control annotations to Handel-C. In *Communication Process Architecture 2007*, Guildford, Surrey, U.K, July 2007. Department of Computing at University of Surrey.

[LNN99] Gerald Lttgen, V. Natarajan, and Rance Cleaveland Gerald L Uttgen V. Natarajan. Priority in process algebras, November 24 1999.

[Low93] G. Lowe. *Probabilities and Priorities in Timed CSP*. PhD thesis, Oxford University, 1993.

[Mat07] Matt. Handel-c forum. http://www.doc.ic.ac.uk/~akf/handel-c/cgi-bin/forum.cgi?msg=956, 2007.

[MWS78] Taylor R. J. B. May, M. D. and C. Whitby-Strevens. Epl: an experimental language for distributed computing. *Trends and Applications: Distributed Processing*, pages 69–71, 1978.

[oai06] *Unifying theories in ProofPower-Z*. SPRINGER-VERLAG BERLIN, February 2006.

[OCW07] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for circus. *Electr. Notes Theor. Comput. Sci*, 187:107–123, 2007.

[Oli05] Marcel Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. Ph.d. thesis, University of York, York, UK, 2005.

[PAW12] G. Lowe J.Ouaknine H. Palikareva A.W. Roscoe P. Armstrong, M.H. Goldsmith and J.B. Worrell. Recent developments in fdr. In *To appear in the proceedings of CAV*, 2012.

[PS] Jonathan D. Phillips and G. S. Stiles.

[PW08] Juan Ignacio Perna and Jim Woodcock. UTP semantics for handel-C. In Andrew Butterfield, editor, *UTP*, volume 5713 of *Lecture Notes in Computer Science*, pages 142–160. Springer, 2008.

[RGGR] Bart Rem, Ajeesh Gopalakrishnan, Tom J. H. Geelen, and Herman Roebbers. Automatic Handel-C Generation from MATLAB and Simulink for Motion Control with an FPGA.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, New York, 1998.

[RR96] A. W. Roscoe and G.M. Reed. The timed failures-stability model for Timed CSP. Technical Report PRG-119, Oxford University Computing Laboratory, 1996. also appeared in Theoretical Computer Science, Vol 211 (1999).

[Sal03] Ashraf Salem. Formal semantics of synchronous systemC. In *DATE*, pages 10376–10381. IEEE Computer Society, 2003.

[SG08] Jens B. Schmitt and Nicos Gollan. Worst case modelling of wireless sensor networks. *Praxis der Informationsverarbeitung und Kommunikation*, 31(1):12–16, 2008.

[SH03] Adnan Sherif and Jifeng He. Towards a time model for *circus*. *Lecture Notes in Computer Science*, 2495:613–??, 2003.

[She06] Adnan Sherif. *A Framework for Specification and Validation of Real Time Systems using Circus Action*. Ph.d. thesis, Universidade Federale de Pernambuco, Recife, Brazil, Jan 2006.

[SJO+05] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[SLD08] Jun Sun, Yang Liu, and Jin Song Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *ISoLA*, pages 307–322, 2008.

[SLM+09] Oliver Sharma, Jonathan Lewis, Alice Miller, Alan Dearle, Dharini Balasubramaniam, Ronald Morrison, and Joe Sventek. Towards verifying correctness of wireless sensor network applications using insense and spin. In Corina S. Pasareanu, editor, *SPIN*, volume 5578 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2009.

[Spi92] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[SS97] Christian Suttner and Geoff Sutcliffe. The TPTP problem library: TPTP v2.0.0. Technical Report AR-97-01, Institut fur Informatik, TU Munchen, 1997.

[Ste05] Jennifer Stephenson. Design guidelines for optimal results in fpgas. http://www.altera.com/literature/cp/fpgas-optimal-results-396.pdf, 2005.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[YBB09] Gang Yao, Giorgio C. Buttazzo, and Marko Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *RTCSA*, pages 351–360. IEEE Computer Society, 2009.