ON THE COMPOSITIONALITY OF ROUND ABSTRACTION

by

MOHAMED NABIH MENAA

A thesis submitted to the
University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
April 2012

UNIVERSITY OF
BIRMINGHAM

University of Birmingham Research Archive

e-theses repository

This unpublished thesis/dissertation is copyright of the author and/or third parties. The intellectual property rights of the author or third parties in respect of this work are as defined by The Copyright Designs and Patents Act 1988 or as modified by any successor legislation.

Any use made of information contained in this thesis/dissertation must be in accordance with that legislation and must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the permission of the copyright holder.

Abstract

Game Semantics is an approach to denotational semantics that has been successful in providing accurate, *fully abstract* models for various programming languages. It has thereafter been applied, amongst other things, to model checking, access control analysis, information flow analysis, and recently, hardware synthesis.

While the roots of modern Game Semantics are sequential, several game models of asynchronous concurrency have since been devised. However, synchronous concurrency has not been considered hitherto.

This thesis studies synchronous concurrency in game-like models. The central idea is to investigate deriving such synchronous models from their asynchronous counterparts using *round abstraction*—a technique that allows aggregating a sequence of computational steps to form a larger, more abstract macro-step.

We define round abstraction within a trace-semantic setting that generalises game semantic models. We note that, in general, round abstraction is not compositional. We then identify sufficient conditions to guarantee correct composition, thereby proposing a framework for round abstraction that is sound when applied to synchronous and asynchronous behaviours.

We explore extensions of our synchronous model with causality, global clocks and determinism.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor, Dan Ghica; his confidence in my ideas, in addition to his support, guidance and advice have had a profound influence on the development of my research.

I drew much motivation and inspiration from many researchers I met during my time as a doctoral student. I am especially grateful to the following people for invaluable discussions and comments about my research: Samson Abramsky, Pierre Clairambault, Olaf Klinke, Pasquale Malacaria, Louis Mandel, Michael Mendler, Andrzej Murawski, Alan Mycroft, Prakash Panangaden, Roly Perera, Uday Reddy, Moshe Vardi and Walter Vogler.

I thank my examiners, Paul Levy and Glynn Winskel, for reading this work with care and offering many useful suggestions.

Thanks to my colleagues at Birmingham—in particular, my officemates in CS 125 and G4 in Chem West—I have had a happy and productive time. Elsewhere, I thank Adel, Karim, Soumia, Mohamed T. and Yasser for their kindness and friendship over the years.

It is hard to accurately express how much I owe to my parents for their support in all I undertook, my brothers for endless entertainment, and my wife Amaria for everything. To them, I dedicate this work.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Outline and Contributions . . . 3

1.3 Publications . . . 5

2 Background **7**

2.1 Synchrony and Asynchrony . . . 8

2.1.1 Asynchrony . . . 9

2.1.1.1 Communicating Sequential Processes . . . 9

2.1.2 Synchrony . . . 12

2.1.2.1 Synchronous Languages . . . 13

2.1.3 Relating Synchrony and Asynchrony . . . 16

2.2 Game Semantics . . . 17

2.2.1 Introduction to Game Semantics . . . 18

2.2.2 Concurrency in Game Semantics . . . 24

2.3 Hardware Compilation via Geometry of Synthesis . . . 26

2.3.1 Introduction to GoS . . . 27

2.3.2 Basic Syntactic Control of Interference . . . 28

2.3.3 A Category of Handshake Circuits . . . 31

2.3.4 A Game-Like Semantics for BSCI . . . 33

2.4 The Problem in Essence . . . 38

2.4.1 Methodology . . . 40

3 A Trace Model of Processes **41**

3.1 Traces and Processes . . . 41

3.1.1 Signatures . . . 42

3.1.2 Traces . . . 43

3.1.3 Processes . . . 45

3.2 Categorical Structure . . . 46

3.2.1 Monoidal Structure . . . 57

3.2.2 Closed Structure . . . 61

4 Compositional Round Abstraction **67**
4.1 Alur-Henzinger Round Abstraction 67
4.2 Round Abstraction on Processes 70
4.3 Compositionality of Partial Round Abstraction 72
4.4 Total Round Abstraction 80
4.5 Discussion 82

5 Causal Processes and Asynchronous Processes **85**
5.1 Justified Traces and Causal Processes 85
5.1.1 Signatures 85
5.1.2 Traces 87
5.1.3 Processes 88
5.1.4 A Category of Synchronous Causal Processes 89
5.2 Asynchronous Processes 100
5.2.1 A Category of Asynchronous Processes 106
5.3 Round Abstraction on Causal Processes 124
5.4 Discussion 128

6 Global Clocks and Determinism **131**
6.1 Clock Monad and Clock Monoid 131
6.2 Clocked Processes 153
6.3 Observable Determinism 163
6.4 Discussion 169

7 Conclusion **173**
7.1 Summary 173
7.2 Further Directions 175

List of Figures

2.1 A landscape of relevant literature . . . 7
2.2 A subset of CSP syntax . . . 10
2.3 The trace semantics of CSP . . . 11
2.4 Asynchronous product versus synchronous product of automata . . . 12
2.5 Some Esterel statements . . . 14
2.6 A cyclic Esterel program . . . 15
2.7 The arena of natural numbers **nat** . . . 20
2.8 The product arena of A and B . . . 21
2.9 The function arena of A and B . . . 21
2.10 The addition of 5 and 7 produced by composition . . . 24

3.1 Coherence condition for α . . . 61
3.2 Coherence condition for λ and ρ . . . 61
3.3 Coherence conditions for γ . . . 62

4.1 A *Counter* in Reactive Modules . . . 69
4.2 A round abstracted *Counter* . . . 69
4.3 Round abstraction on automata . . . 70
4.4 When compatibility is too strong . . . 74
4.5 Totality of $\preceq_u$ in Lemma 4.10 . . . 78

5.1 Lemma 5.16, Case 2 . . . 93
5.2 Lemma 5.16, Case 3 . . . 93
5.3 Pointer structure inheritance in Lemma 5.20 . . . 96
5.4 Causality in $id_{A \otimes B}$. . . . 98

6.1 Clock monad . . . 134
6.2 Justification pointers in traces $u \in \alpha_{Ck,Ck,Ck}; id_{Ck} \otimes sp_{Ck}; sp_{Ck}$ in Lemma 6.4 . . . 142
6.3 Justification pointers in traces $v \in sp_{Ck} \otimes id_{Ck}; sp_{Ck}$ in Lemma 6.4 . . . 142
6.4 Justification pointers in traces $u \in cs_{Ck} \otimes id_{Ck}; sp_{Ck}$ in Lemma 6.4 . . . 142
6.5 Justification pointers in traces $v \in \lambda_{Ck}$ in Lemma 6.4 . . . 142

6.6 A diagrammatic representation of the monoid axioms 142
6.7 A circuit representation of the double strength map. 151
6.8 A circuit representation of composition in the Kleisli category $\mathbf{SynProc}_T$. 152
6.9 A circuit representation of $(id_B \otimes_T \tau) ;_T eval_{B,C_T}$. 152

CHAPTER 1

INTRODUCTION

Abstractions are everywhere. Each instant, our brains ignore millions of stimuli, selectively processing those deemed relevant; our very perception of the world is an abstraction [SCME06].

In computer science, our study of systems is guided by formal models that abstract away from the physical world. Process calculi conveniently model *observable* events, an idealisation of reality. Models of synchronous languages assume an infinitely powerful machine that allows instantaneous computation. Automata use discrete time steps.

This thesis examines the use of a form of temporal abstraction in correlating two distinct views of concurrent behaviour—synchronous and asynchronous—*compositionally* and thereby lays the foundations for a synchronous formulation of Game Semantics.

1.1 Motivation

One taxonomy for concurrency theory classifies its models into two broad categories: synchronous and asynchronous. In general, synchronous systems are those that synchronise their operation on a clock, be it global or local. In contrast, asynchronous systems are those that lack this feature, and therefore, whose components proceed independently according to a set protocol.

In this thesis, the fulcrum of this distinction lies in the presence or absence of the notion of *simultaneity*. Hence, the key difference between the two is that in a synchronous setting, we must consider the case of two events occurring simultaneously whereas in the asynchronous case, it is impossible to ascertain that. This view is common in process calculi, for example, where the failure of simultaneity is expressly stated.

> *"We can validly ignore the possibility that two events occur simultaneously; for if they did, the observer would still have to record one of them first and then the other, and the order in which he records them would not matter."* [Hoa85]

Consequently, application areas for the two paradigms have typically been different: asynchronous concurrency is used when bounds on the time necessary for interaction cannot be guaranteed (e.g. distributed systems), or when time is intentionally abstracted (e.g. high-level programming languages), whereas synchronous concurrency is commonly used when time is an essential facet of the system (e.g. safety-critical systems).

The correlation of synchrony and asynchrony has been an object of research for a long time. Starting with Milner's seminal work in the 1980s, several authors demonstrated how asynchronous models can be derived from [Mil83, BCG99] or simulated by [HM06] their synchronous counterparts. Hence, it is commonly accepted that synchronous models are more expressive. The opposite direction, from asynchrony to synchrony, has received less attention. So, while the naive synchronous representation of an asynchronous process is inefficient, deriving a *low-latency* representation is arguably challenging. Even recovering synchronicity after it is removed from a specification is a non-trivial procedure [BCG99].

In their specification language, *Reactive Modules*, Alur and Henzinger take a more general approach [AH99]. In particular, they describe a technique, which they name *round abstraction*, that allows arbitrarily many computational steps to be aggregated into a single macro-step. This forms the basis for an elegant solution to the problem of building synchronous systems from asynchronous specifications. We consider round abstraction, essentially, as an approximation technique that removes some of the timing information between events in a process.

One area in which the correlation between synchrony and asynchrony has yet to be investigated is Game Semantics [HO00, AJM00]. This denotational *intensional* semantics has met with success in the 1990s when it led to fully abstract models for the prototypical language PCF, a challenge that was outlined nearly two decades earlier [Plo77]. Several asynchronous concurrent game models have subsequently appeared in the literature, e.g. [Lai01b, Lai06, GM08]; but no synchronous homologue as yet.

This thesis studies the compositionality of round abstraction in causal and non-causal trace

models, and thereby lays the foundations for the formulation of a synchronous game semantics. We formulate round abstraction within a game-like locally synchronous model that can accommodate synchronous and asynchronous behaviours. We then study the compositionality of round abstraction, a question unanswered in the original work. We also explore how round abstracted models can be extended with global synchrony and determinism.

1.2 Outline and Contributions

The thesis is structured as follows.

1 – Introduction.

2 – Background. This chapter explores concepts that underpin the topic of this thesis, including synchronous and asynchronous concurrency, Game Semantics, and hardware compilation using Geometry of Synthesis. We discuss relevant literature, placing the thesis in the context of its broader influences, and finish with a more elaborated statement of the problem addressed herein.

3 – A Trace Model of Processes. We introduce the details of the setting in which the problem will be studied. We extend the notion of a trace to allow recording simultaneous events. Subsequently, we describe a trace model of low-level concurrency, which we show to have a closed symmetric monoidal structure.

4 – Compositional Round Abstraction. After reviewing round abstraction in its original formulation, we define a compositional form thereof within the setting outlined in Chapter 3. We introduce two notions of round abstraction on processes: *partial* and *total*, which represent different levels of accuracy in the abstraction. So, while partial round abstraction only requires that all traces in the abstraction approximate those in the original process, its total homologue additionally states that all traces in the original process be abstracted. We show that, in general, neither notion is compositional. We then proceed to identifying sufficient conditions that guarantee compositionality of partial round abstractions. We discuss total round abstraction as an open problem.

Contribution. The original notion of round abstraction applies to whole systems and does not address the question of whether round abstracted systems interact correctly with each other. We remedy this by introducing the first compositional formulation of round abstraction.

5 – Causal Processes and Asynchronous Processes. We augment the trace model of Chapter 3 with causality in the form of justification pointers and obtain a closed symmetric monoidal category. Then, we introduce a category of interleaved asynchronous processes, in the style of Ghica and Murawski's concurrent game model [GM08]. We prove that partial round abstraction is compositional in both the causal and asynchronous settings.

Contribution. We prove that partial round abstraction is compositional on causal and asynchronous processes. The conditions guaranteeing compositionality are the same ones used for non-causal processes. New alternative formulations of copycat and asynchronous identities are introduced alongside new categorical proofs, cf. [GM08].

6 – Global Clocks and Determinism. We demonstrate that locally synchronous processes can be extended with a global clock in a principled fashion. Following an established tradition, we use a computational monad. The monad describes how globally clocked processes should be wired. We then describe a category of clocked processes, where events are globally synchronised with the clock. We use this globally synchronous category to study determinism. We find that deterministic processes can be defined when justification pointers encode sufficient causality information.

Contribution. While computational monads are, by now, an established technique to enrich categorical models with new features, the idea of a clock monad is, to our knowledge, novel. We define a category of globally clocked processes. We discover that subtle problems in characterising deterministic processes in the synchronous setting arise when we rely on justification pointers as a sufficient notion of causality.

7 – Conclusion. We conclude by summarising our results and suggesting potential avenues for further research.

1.3 Publications

This thesis is partly based upon the following publications.

[GM11] D. R. Ghica and M. N. Menaa. Synchronous game semantics via round abstraction. In M. Hofmann, editor, *FOSSACS '11: Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures (Saarbrücken, Germany)*, volume 6604 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2011.

[GM10] D. R. Ghica and M. N. Menaa. On the compositionality of round abstraction. In P. Gastin and F. Laroussinie, editors, *CONCUR '10: Proceedings of the 21th International Conference on Concurrency Theory (Paris, France)*, volume 6269 of *Lecture Notes in Computer Science*, pages 417–431, 2010.

In addition, this thesis is partly based on the following unpublished work.

[GM] D. R. Ghica and M. N. Menaa. Low-latency synchronous representations of asynchronous processes. Submitted to the *Journal of the ACM*.

[Men10] M. N. Menaa. On the compositionality of round abstraction. Short paper presented at the 25th Annual Symposium on Logic in Computer Science (LICS '10), Edinburgh, Scotland, UK, 2010.

[Men09] M. N. Menaa. Towards a synchronous game semantics. Slides presented at the 4th Workshop on Games for Logic and Programming Languages (GaLoP IV), York, UK, 2009.

CHAPTER 2

BACKGROUND

We begin by examining some topics that will help situate this thesis in the context of its broader influences. The original motivation for our work stems from Geometry of Synthesis [Ghi07], a hardware compilation technique from higher-order programming languages to digital circuits via Game Semantics. While compiling to asynchronous hardware has been previously demonstrated [GS11], targeting synchronous hardware presents new challenges stemming from the discrepancy between the asynchronous input of the compiler and the desired synchronous output.

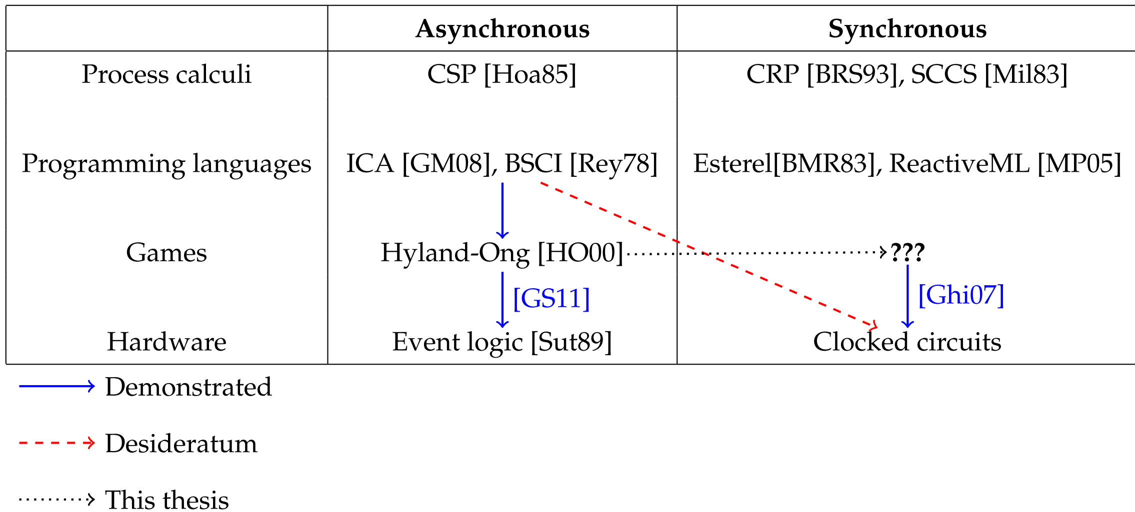


Figure 2.1: A landscape of relevant literature

We first contrast synchronous and asynchronous forms of concurrency, taking Communicating Sequential Processes [Hoa85] and Esterel [BMR83] as prime examples. We then briefly introduce the main notions of Game Semantics [HO00], focusing on the concurrent formulation due to Ghica and Murawski [GM08]. Finally, we review hardware compilation using Geome-

try of Synthesis. These topics are only examined insofar as the subsequent exposition requires; the following sections are by no means representative of their topics. References for further reading are provided throughout.

2.1 Synchrony and Asynchrony

In early days, computers were essentially viewed as sequential machines. This useful abstraction led to the development of a rich theory—including Turing machines and the lambda calculus—which in turn had a deep impact on subsequent developments in computing. Issues of parallel computation were primarily motivated by operating system design, for example, Dijkstra's semaphores [Dij65]. However, the advent of computer networks in the 1970s dictated the need for a different class of *concurrent* models. In contrast to sequential models, where processes only communicate when they terminate, concurrent ones consist of independent sequential components proceeding in parallel *while* communicating.

Later developments yielded two broad classes of concurrent models: synchronous and asynchronous, depending on the nature of communication between concurrent processes. The key distinction between the two is that in the former, one must consider the case of two or more events occurring *simultaneously* whereas in the latter, it is impossible to ascertain that.

It is perhaps due to this convenient abstraction that asynchronous concurrency developed rapidly. Petri Nets [Pet62] were among the first and most influential models of concurrency and are still studied to this day. Other prominent examples include Mazurkiewicz traces [Maz77] and Winskel's event structures [Win87]. These three models share a common trait that separates them from others: they all model concurrency in an explicit and so called *non-interleaved* way. In other words, the internal components of a concurrent system can be discerned.

On the opposite side of the spectrum, process calculi like the Calculus of Communicating Systems (CCS) [Mil80] and Communicating Sequential Processes (CSP) [Hoa85] together with their semantic models, respectively synchronisation trees [Mil80] and Hoare traces [Hoa80] are *interleaved* models. These hide their internal structure and describe computation as sequential observations of the aggregate behaviour of their components. As a result, they have often been criticised for *reducing concurrency to nondeterminism*.

On the other side of the fence, synchronous concurrency appeared in the beginning of the 1980s with Milner's seminal work on the Synchronous Calculus of Communicating Systems (SCCS) [Mil83] and Austry and Boudol's MEIJE calculus [AB84, Bou85].

Around the same time, a somewhat different approach to synchronous concurrency was taken by a community of scientists motivated by real-time systems. Seminal papers on Esterel [BMR83], Statecharts [HP85], Lustre [BCH$^+$85] and SIGNAL [LGBBG86] marked the inception of synchronous programming languages.

2.1.1 Asynchrony

A full review of asynchronous models of concurrency is beyond the scope of this thesis. A good survey of some of these models using a categorical approach can be found in [WN95]. We will instead give an overview of one of the pioneering (asynchronous) process calculi, Communicating Sequential Processes (CSP), and its trace semantic model [Hoa80], often called Hoare traces. These share our approach to asynchronous concurrency as described in Chapter 3: linear time and interleaved. Moreover, CSP is an important precursor to Game Semantics and therefore, serves as a gentle introduction to it. This idea was aptly described by Abramsky in a recent essay [Abr10].

There are many good CSP references, for example [Hoa85, Ros98, Sch00]. For a historical overview of the development of process calculi, including CSP see [Bae05].

2.1.1.1 Communicating Sequential Processes

The basic element in process calculi is called a *process*. It is an idealised description of the behavioural pattern of a system. Behaviour is anything that can be *observed* and idealisation implies that a process is often an abstraction of the *real* behaviour. In the case of CSP, a process is described by *events*, which are instantaneous atomic observable actions that a system can exhibit. The set of all events occurring in the process description, denoted by αP for a process P, is called its *alphabet*. The possibility that two distinct events occur simultaneously is always ignored, and whenever two processes synchronise on the same event, it is only recorded once*.

*This limited form of synchrony is often called rendezvous or handshake communication.

$$
\begin{array}{llll}
P & ::= & STOP & \text{Deadlocking process} \\
 & | & SKIP & \text{Terminating process} \\
 & | & x \rightarrow P & \text{Prefixing} \\
 & | & P \setminus X & \text{Hiding} \\
 & | & P \mathbin{\Box} P & \text{External choice} \\
 & | & P \sqcap P & \text{Nondeterministic choice} \\
 & | & P \parallel P & \text{Parallel composition} \\
 & | & P \mathbin{|||} P & \text{Interleaving} \\
 & | & P \mathbin{;} P & \text{Sequential composition}
\end{array}
$$

Figure 2.2: A subset of CSP syntax

CSP offers a set of algebraic primitives to build more complex processes from basic ones. Figure 2.2 shows a subset of CSP syntax. The primitive processes considered here are $STOP$ and $SKIP$ which model deadlock and termination respectively. The process $x \rightarrow P$ is one which partakes in the event x and behaves like P afterwards. In the process $P \setminus X$, every event in the set X is removed from the description of P. This is called hiding and corresponds to internalising a subset of the interface of a process so that it is no longer visible to its environment. Whereas the process $P \mathbin{\Box} Q$ lets the environment choose whether it subsequently behaves like P or like Q, this choice is made internally and nondeterministically in $P \sqcap Q$. CSP allows two main concurrency constructs: interleaving, where processes proceed autonomously; and parallel composition, where processes have to synchronise on common events.

Every primitive in CSP has rules governing how processes involving it can be rewritten without altering their meaning. Hence, some of these laws induce a simple notion of syntactic equivalence. For example, parallel composition is both symmetric and associative. Therefore, $P \parallel (Q \parallel R) = (Q \parallel P) \parallel R$. Additionally, some of these laws allow us to simplify a process expression. For instance, $P \parallel STOP = STOP$ and $SKIP \mathbin{;} P = P$

We will now look at the trace model of CSP. The basic idea is to model the behaviour of a process using traces: sequences of events over its alphabet. An often used metaphor is that of a neutral observer writing the events of a process in a notebook as they occur. The semantics of the process, therefore, is the collection of all its possible traces. This naturally yields a notion of process equivalence: processes are *trace equivalent* when their respective sets of traces are equal. Along with its simplicity, trace equivalence is known to be the least discriminating process equivalence. For example, although $P \mathbin{\Box} Q$ is trace equivalent to $P \sqcap Q$, these two processes

will generally not behave the same when interacting with their environment. CSP has been given several other semantic models that yield increasingly discriminating notions of process equivalence.

First, we need to introduce some notation. The empty trace is written ϵ. If s and t are traces, we write their concatenation as $s \cdot t$. Moreover, if s is a trace in P and $X \subseteq \alpha P$, then $s \upharpoonright X$ is the trace obtained by deleting all non members of X from s. For the sake of uniformity, we use the same notation for similar notions introduced in later chapters. This does not necessarily comply with the original notation of CSP [Hoa85].

For any CSP process P, its set of traces, $\mathsf{traces}(P)$, is

- nonempty: it always contains the empty trace ϵ which corresponds to the behaviour of the process up to the point where it engages in its first event,
- *prefix-closed*: if $s \cdot t$ is a trace of P, then s is also a trace of P. This models the history of execution: if we can observe the behaviour $s \cdot t$, we must have previously observed the behaviour s.

The trace semantics of CSP is defined inductively over its syntax by rules for each operator and primitive. In Figure 2.3, we describe those corresponding to the syntax in Figure 2.2. Successful termination is denoted by the reserved symbol ✓.

Note that these definitions result in a *compositional* semantics: the meaning of a process is a function of the meanings of its components.

$$
\begin{array}{lcl}
\mathsf{traces}(STOP) & = & \{\epsilon\} \\
\mathsf{traces}(SKIP) & = & \{\epsilon, \checkmark\} \\
\mathsf{traces}(a \rightarrow P) & = & \{a \cdot s \mid s \in \mathsf{traces}(P)\} \\
\mathsf{traces}(P \setminus X) & = & \{s \upharpoonright (\alpha P - X) \mid s \in \mathsf{traces}(P)\} \\
\mathsf{traces}(P \,\Box\, Q) & = & \mathsf{traces}(P) \cup \mathsf{traces}(Q) \\
\mathsf{traces}(P \sqcap Q) & = & \mathsf{traces}(P) \cup \mathsf{traces}(Q) \\
\mathsf{traces}(P \parallel Q) & = & \mathsf{traces}(P) \cap \mathsf{traces}(Q) \\
\mathsf{traces}(P \,|||\, Q) & = & \{s \mid s \upharpoonright \alpha P \in \mathsf{traces}(P) \text{ and } s \upharpoonright \alpha Q \in \mathsf{traces}(Q)\} \\
\mathsf{traces}(P\,;Q) & = & \mathsf{traces}(P) \cup \{s \cdot \checkmark \cdot t \mid s \cdot \checkmark \in \mathsf{traces}(P) \text{ and } t \in \mathsf{traces}(Q)\}
\end{array}
$$

Figure 2.3: The trace semantics of CSP

2.1.2 Synchrony

The synchronous paradigm has a long pedigree in hardware design. Synchronous digital circuits synchronise their operation using a global clock that sets a strict upper bound on when they must stabilise. This results in an abstraction whereby these circuits proceed in consecutive discrete steps. All events occurring during one such step are considered to be simultaneous.

The same idea eventually found its way to concurrency theory. Milner's Synchronous Calculus of Communicating Systems (SCCS) [Mil83] was the first notable work in this direction. It models simultaneity using an Abelian semigroup of actions as follows.

Agents are drawn from a class **P**. Moreover, there is a set **Act** of *atomic* actions: time is assumed to be discrete and actions indivisible in time. For each action $a \in \mathbf{Act}$ a transition relation is defined over **P**, where

$$P \xrightarrow{a} P'$$

indicates that the agent P may perform a and subsequently become P'. The class of agents is quotiented by bisimulation (a notion of process equivalence) to yield processes.

The product of the Abelian semigroup is used as follows. If $P \xrightarrow{a} P'$ and $Q \xrightarrow{b} Q'$, then $P \times Q \xrightarrow{ab} P' \times Q'$. That is, the system consisting of P and Q may perform a and b simultaneously and transform into the system consisting of P' and Q'. Figure 2.4, adapted from [Cas01], contrasts this synchronous product with its asynchronous equivalent. Idleness can be modelled using a reserved action '1' and further requires that $(\mathbf{Act}, \times, 1)$ is an Abelian monoid.

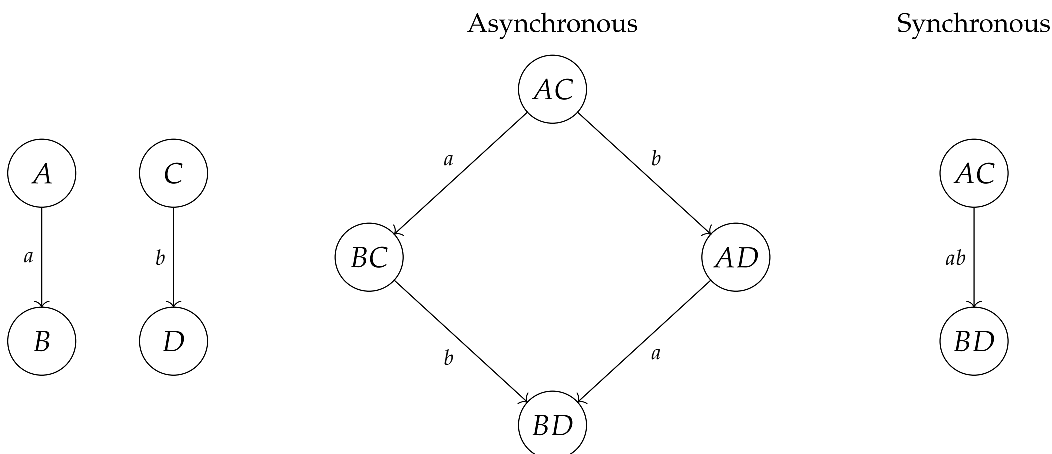


Figure 2.4: Asynchronous product versus synchronous product of automata

This is important because it allows SCCS to model asynchronous behaviour through *stuttering*. A process is said to stutter if it may nondeterministically 'do nothing'.

The idea of using a monoid of actions as a basis for a synchronous calculus was further explored by Austry and Boudol in their MEIJE calculus [AB84, Bou85]. See [De 85] for a joint presentation.

SCCS and MEIJE paved the way for a family of programming and specification languages called *synchronous languages*.

2.1.2.1 Synchronous Languages

Synchronous languages combine synchronous concurrency (*à la* SCCS) and determinism. They assume *the synchronous hypothesis*, an idealisation stipulating that, on some level of abstraction, processes compute and communicate in *zero time*. It is therefore left to the implementation to correctly approximate the synchronous assumptions [PBEB07].

According to Benveniste and Berry, systems under the synchronous hypothesis *"compose very well and turn out to be easier to describe and analyze than asynchronous ones"* [BB91]. Furthermore, it facilitates deterministic concurrency. This is a desirable and often compulsory feature in *reactive* systems [HP85, Hal93], which are often required to guarantee the same behaviour for any input.

Another assumption in synchronous languages is the discretisation of time into *instants*. Hence, as in SCCS, programs execute in successive atomic *reactions*.

In the following, we will succinctly review Esterel [BMR83], an imperative synchronous programming language. Since its inception, it has been industrially adopted in avionics and wireless communication hardware, and more generally in reactive embedded systems.

An Esterel program typically consists of several *modules*. Each module has a *header* and a *reactive body*. The header declares the name of the module and its interface. The body consists of concurrent threads that execute with respect to a global clock, whose ticks delineate successive instants. During any instant, each thread executes independently, and either terminates or halts to resume in the next instant. Communication across threads is achieved using *signals*: globally visible events that are instantaneously broadcast.

`nothing`	no-op
`pause`	hold until the next instant
p `;` q	sequential composition
$p \parallel q$	parallel composition
`loop` p `end`	infinite iteration
`emit` S	emit signal S
`present` S `then` p `else` q `end`	signal branching
`suspend` p `when` S	pre-emption test
`if` v `then` p `else` q `end`	conventional branching

Figure 2.5: Some Esterel statements

There are no global declarations of signals or variables, and therefore, modules exclusively communicate using the signals declared in their headers. A single module called the *root module* instantiates other modules, which in turn, can instantiate further modules using the `run` command. Nested module declarations and recursive instantiation are not permitted. The `run` statement is a form of *inlining*: instantiating a module instructs the compiler to replace the `run` statement in the *caller module* by the reactive body of the *instantiated module* with the appropriate instantiation of formal parameters [PBEB07]. Observe that Esterel does not support functions.

In every instant, executing reactive code amounts to evaluating the status and data value of each signal in a deterministic way. This proceeds in two ways. First, at the start of each instant, all signals are in an 'undetermined' state (except input signals) and become present only if an `emit` statement is executed. Second, as soon as the emission of a signal is deemed impossible by the control flow, it is set to absent. It is easy to see the possibility for the presence of a signal to depend on itself. The program in Figure 2.6 illustrates this: X depends on Y being present and vice versa.

Instantaneous cycles are identified via *causality* analysis. Intuitively, an event A *causes* another event B if the occurrence or absence of B is sufficient to determine the occurrence or absence of A [PBEB07]. Hence, causality defines an execution order within each reaction. There are three types of causal dependencies: sequencing, signal communication, and access to shared variables. Usually, non-causal program have no semantics and therefore, are rejected by the compiler. This is enforced using a semantics based on *constructive logic* [TvD88]. For more on the theory of causality, refer to [Ber99].

```
module Cycle :
output X,Y
  present X then emit Y end
||
  present Y then emit X end
```

Figure 2.6: A cyclic Esterel program

Other important semantic concepts in Esterel include *reincarnation* and *schizophrenia*. Since most language constructs in Esterel execute instantaneously, it is possible for a loop iteration to end and the following one to begin within the same instant. This may cause a signal to be emitted in one iteration and the loop to restart with a fresh copy in the next. This phenomenon is known as reincarnation. If several *copies* of the same signal have different statuses in the same instant, it is called schizophrenic. Reincarnation and schizophrenia are handled in the semantics in order to preserve the synchronous nature of the language [PBEB07]. However, alternative methodologies have been proposed [TdS04, SW01] to eliminate this phenomenon altogether.

The family of synchronous languages also includes Lustre [HCRP91] and Signal [LLGL91]. See [BB91] for a survey circa 1991. More recent ones appear in [Hal98] and [BCE+03].

Finally, we note that similar ideas have independently appeared elsewhere in computer science. For instance, *Bulk Synchronous Parallel* (BSP) [Val90] is a parallel programming model that shares some of the basic ideas of synchronous languages. It prescribes independent sequential execution of processes that are connected by a communication mechanism, and which synchronise globally. BSP programs execute in a sequence of *super-steps*, each composed of three phases:

1. Each process executes its local sequential code. It can request data from other (remote) processes.
2. The environment (network) delivers the requested data.
3. A global synchronisation event occurs, therefore making the data available at the start of the next super-step.

Languages implementing BSP include Bulk Synchronous Parallel ML (BSML) [LHF00, LGG+08] and Minimally Synchronous Parallel ML [LGAD04].

2.1.3 Relating Synchrony and Asynchrony

A focal point in theoretical computing is the study of correlations between models of computation. For example, in [NSW93], Nielsen, Sassone and Winskel establish formal relationships between some well-known models of asynchronous concurrency, allowing to *translate* between them.

Given the aforementioned synchronous/asynchronous dichotomy, an interesting question concerning the correlation of the synchronous and asynchronous paradigms arises: can we derive one from the other?

Milner was the first to establish that asynchronous computation can be modelled using a synchronous calculus (SCCS) [Mil83], also showing how the asynchronous Calculus of Communicating Systems (CCS) [Mil80] can be derived from SCCS by allowing processes to *stutter*. Later work showed similar results in varied contexts. For example, in [HB02] and [HM06], the authors show that synchronous Mealy machines can model asynchronous behaviour by introducing stuttering through sporadic activation and nondeterminism through additional arbitrary inputs (also called oracles).

Mixing synchrony and asynchrony has also received considerable attention. The Globally Asynchronous Locally Synchronous paradigm [Cha84] has been proposed to employ synchronous modules in an asynchronous environment. Several approaches were suggested to map synchronous systems into this framework, for instance, [Ben01, PBC07]. Related approaches include Communicating Reactive Processes [BRS93], in which synchronous modules communicate via CSP rendezvous and Multiclock Esterel [BS01], which substitutes local clocks for a global one.

Deriving synchronous formalisms from asynchronous ones has not received as much attention; perhaps due to a lack of practical applications to motivate it. It is known that a naive synchronous representation of asynchronous processes can be achieved by limiting the number of simultaneous events to one. However, this is unsuitable if what is desired is not just a correct encoding, but one that additionally has a *low-latency*. This is not easy; even recovering synchronicity after it is removed from a specification is a non-trivial procedure [BCG99].

A first step in this direction was afforded by the specification language Reactive Mod-

ules [AH99]. In addition to combining synchronous and asynchronous primitives, it provides a technique, called *round abstraction*, allowing synchronous specifications to be constructed from asynchronous ones. Round abstraction is a central notion in our work, and is introduced in some detail in Chapter 4.

2.2 Game Semantics

Contrary to what may appear upon first encountering the term, Game Semantics is neither a misnomer nor an oxymoron. To understand the 'game' metaphor and how it came to be used for 'semantics', one has to delve into its history.

Immediate precursors of Game Semantics, *dialogue games*, have been known since Aristotle [Ari28] and studied throughout history [Ham70], in a mostly informal fashion. For instance, medieval literature describes so called *obligationes*, a sort of debate challenge involving two protagonists: a *Respondens* and an *Opponens*. Assuming the truth of a typically false statement, the aim of the former is to give rational answers to questions from the Opponens, while the latter tries to drive him into logical contradiction [Hod04]. In the late 1950s, dialogue games were formalised by Lorenzen [Lor60, Lor61, LL78] and his student Lorenz [Lor68] and proposed as a semantic model for constructive logic.

Lorenzen's idea was not too different from medieval obligationes. He proposed to interpret a logical statement as a two-person game between a *Proponent* and an *Opponent*. While the Proponent defends the truth of the formula the Opponent aims to refute it. A proof of the formula is then equivalent to the existence of a *winning strategy* for the Proponent, in the game-theoretic sense. For a historical perspective on the inception of dialogue games as a model for logic see [Lor01].

Another breakthrough came in 1976. While investigating how numbers arise from a certain class of games, Conway [Con76] laid the foundation for a compositional treatment of games. Indeed, he introduced the crucial notion of playing games in parallel. Building on his work, Joyal [Joy77] formulated the first category of games and strategies by taking the sum in Conway Games as the tensor product, which results in a compact closed structure.

In 1992, Blass [Bla92] produced the first game semantic model for Girard's linear logic [Gir87].

This was shortly followed by a paper by Abramsky and Jagadeesan [AJ92, AJ94] which overcame some technical problems in Blass's work and gave a fully complete model of the unit-free multiplicative fragment of linear logic. While Blass [Bla92] had already alluded to the idea, Abramsky and Jagadeesan's work had the additional merit of drawing explicit parallels between game semantic notions, used until then to model logic, and computational processes [Hoa85, Mil80]. This analogy together with the full completeness result were significant, as they foreshadowed the possibility of using Game Semantics to tackle the long-standing problem of full abstraction for PCF, which remained unsolved for over 15 years.

First formulated in the work of Milner [Mil75] and Plotkin [Plo77], the full abstraction problem can be roughly understood as finding a syntax free model for a programming language, that is *as precise as possible*, or in Milner's words not *"over-generous"*. This requires for any two objects in the model to be the same precisely when their responding terms in the language behave likewise. The problem was first formulated for the programming language PCF in [Mil77, Plo77].

In 1993, the problem finally fell at the hands of three independent research teams: Abramsky, Jagadeesan and Malacaria [AJM00]; Hyland and Ong [HO00]; and Nickau [Nic94]. All three fully abstract models used game models. This result marked the inception of modern Game Semantics and sparked interest in modelling programming language features using games. Game models for recursive types [AM94], local state [AM97b], call-by-value [AM97a], polymorphism [Hug97, AJ05], control [Lai97, Lai03], general references [AHM98, Lai07], passive types [AM99a], nondeterminism [HM99], exceptions [Lai01a], and concurrency [Lai01b, Lai06, GM08, Mur10] have been introduced.

Game Semantics has also been used for model checking [AGMO04, DGL06, GM06, BG08, GB09, BG09, BDGL10], access control analysis [AJ09] and information leakage [CKP09], among other applications. It remains an active research area, as demonstrated by recent advances at the foundational level, e.g. [DH01, HL06, Mel06, MM07, CM10].

2.2.1 Introduction to Game Semantics

We informally introduce the salient concepts of Game Semantics in the style of Hyland and Ong [HO00]. In the following exposition, we assume we are working in a call-by-name frame-

work. For further insight, the interested reader is encouraged to consult some of the many good introductory papers in the literature, for example [Abr97a, Abr97b, Hyl97, McC97, AM99b, Abr01, J02, Ghi09a].

As the game metaphor suggests, Game Semantics models computation as a game between two players: a Proponent (P) representing the system (or term) and an Opponent (O) representing its environment (or the context in which the term occurs).

Game Semantics is *denotational* in nature: it uses mathematical objects as a basis for semantics. The objects in question here are *strategies* played on *arenas*. These model terms and types respectively. Arenas are defined by a poset of atomic actions, called *moves* and used to model computational steps. Moves can belong to either the Opponent or the Proponent and can be either *questions* or *answers*. This yields the following taxonomy of moves (adapted from [Abr97b]).

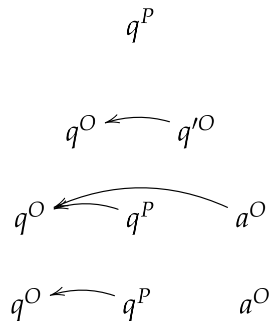
O question	request output from P
O answer	provide input to P
P question	request input from O
P answer	provide output to O

The partial order in arenas formalises the idea that some moves cannot occur unless *enabled*, or *justified*, by moves occurring earlier. Only questions can enable further moves, which can be questions or answers. Furthermore, Opponent moves can only enable Proponent moves, and vice versa.

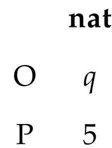
Strategies are sets of traces, called *plays* or *positions*, defined as finite sequences of moves. They model the behaviour of a system by describing how it interacts with its environment, i.e., its *a priori* semantics in the terminology of Francez *et al.* [FHLR79]. Strategies must obey the rules of the game, which depend on the programming language being modelled. For example, for PCF [HO00], the Opponent always starts first and the players take turns playing. Moreover, plays must satisfy the principles of *polite conversation*: a question is only asked if an earlier question warrants it; if several questions are pending, the most recent one is answered first; and no answer should be given to a question that was not asked.

In so called *pointer games*, plays are equipped with *justification pointers* such that each move

occurrence must *point* to an earlier one that can enable it in the arena. For example, the following plays, where q denotes a question and a an answer, are illegal.



With the concepts introduced so far, we can already give interesting examples. The arena **nat** corresponding to natural numbers has the shape depicted in Figure 2.7. Any play of this type begins with the Opponent question q, to which the Proponent answers by producing a natural number. For instance, the constant 5 has the following play.



This style for writing plays is common in Game Semantics, and should be read downwards. Justification pointers are often dropped when they can be inferred.

Given arenas for base types, we can form product and function arenas in a systematic way. The product arena $A \times B$ corresponds to the arenas operating sides by side. The function arena allows B to 'query' A. This requires the O/P polarities of A moves to be reversed, and each tree in B to be connected to its own copies of each tree in A.

As an example, consider the constant $(5, 7)$, which has the following two plays.

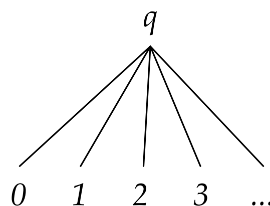


Figure 2.7: The arena of natural numbers **nat**

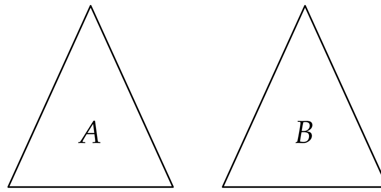


Figure 2.8: The product arena of A and B

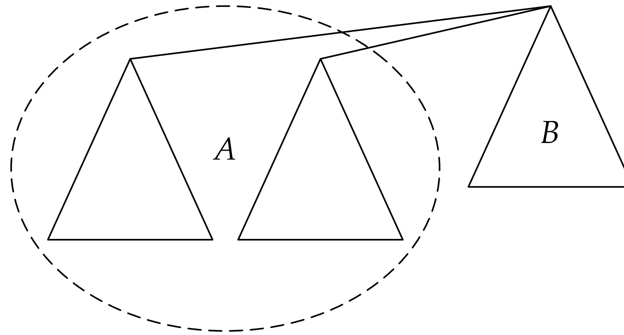


Figure 2.9: The function arena of A and B. Note that B consists of a single tree whereas A has two

	nat	$\times$	**nat**
O	q		
P	5		
O			q
P			7

	nat	$\times$	**nat**
O			q
P			7
O	q		
P	5		

Observe that the evaluation of the pair has no inherent directionality.

Using the function arena, we can model functions. Addition for example has the following typical play.

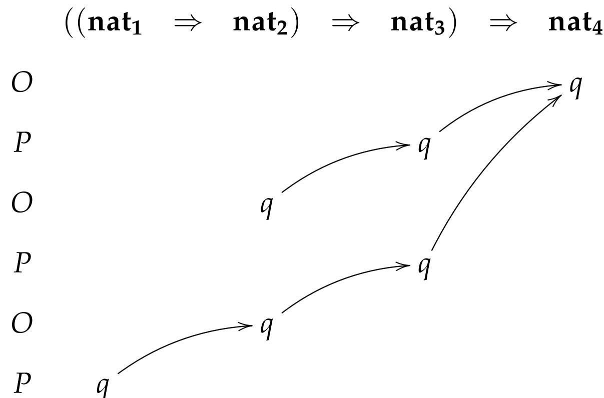
	nat	$\times$	**nat**	$\Rightarrow$	**nat**
O					q
P	q				
O	n				
P			q		
O			m		
P					$n+m$

It starts by the Opponent prompting the addition term to start evaluating, at which point the Proponent requests the first and the second arguments, successively, which are provided by the Opponent. Finally, the Proponent returns the value corresponding to the addition of the two arguments. Note that while the product of two **nat** arenas does not imply directionality, our definition of the addition function forces a left to right evaluation of its arguments.

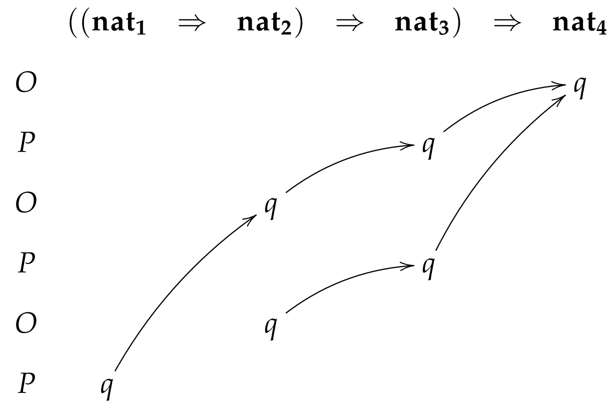
Higher-order functions can also be defined. For example, $\lambda f.f(f(1))$ has the following typical play.

	(**nat**	$\Rightarrow$	**nat**)	$\Rightarrow$	**nat**
O					q
P			q		
O	q				
P			q		
O	q				
P	1				
O			n		
P	n				
O			m		
P					m

Observe that the two plays of **nat** $\Rightarrow$ **nat** are interleaved in this example. At higher-order types, this interleaving can cause ambiguity if justification pointers are not used. This brings us back to importance of justification pointers. Consider the following situation, described in [AM99b]. Let terms M, N be defined as, $M = \lambda f.f(\lambda x.f(\lambda y.y))$ and $N = \lambda f.f(\lambda x.f(\lambda y.x))$ and played on arena $((\mathbf{nat_1} \Rightarrow \mathbf{nat_2}) \Rightarrow \mathbf{nat_3}) \Rightarrow \mathbf{nat_4}$. While M has plays of the form,



the second term, N, has plays of the following shape.

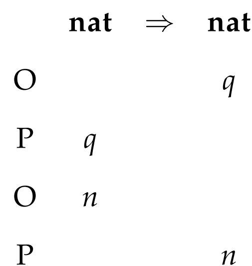


Discarding the pointer structure results in the identification of these two different lambda terms.

In certain game models, justification pointers are dropped, either because situations akin to the previous example do not arise, or when the justification structure is implicit and can be recovered from the structure of the play [GM03].

One of the most important operations in game semantics is *composition*. It describes how two systems interact, hence providing a way to build 'larger' systems from smaller ones. Given two strategies σ, τ on arenas $A \Rightarrow B$ and $B \Rightarrow C$ respectively, we can compute the composite strategy $\sigma;\tau$ on arena $A \Rightarrow C$. Intuitively, this is achieved by synchronising on the moves of the two B arenas (which have complementary O/P polarities), then hiding them. This has often been described as 'parallel composition followed by hiding' in the sense of CSP. Figure 2.10 depicts the composition of the addition strategy with the strategy for the constant $(5, 7)$.

The final notion we examine is that of *copycat strategies*. For each arena A, there exists a strategy on $A \Rightarrow A$, where the Proponent replicates the moves issued by the Opponent in one copy of A, in the other. At arena **nat**, for example, we get the following play.



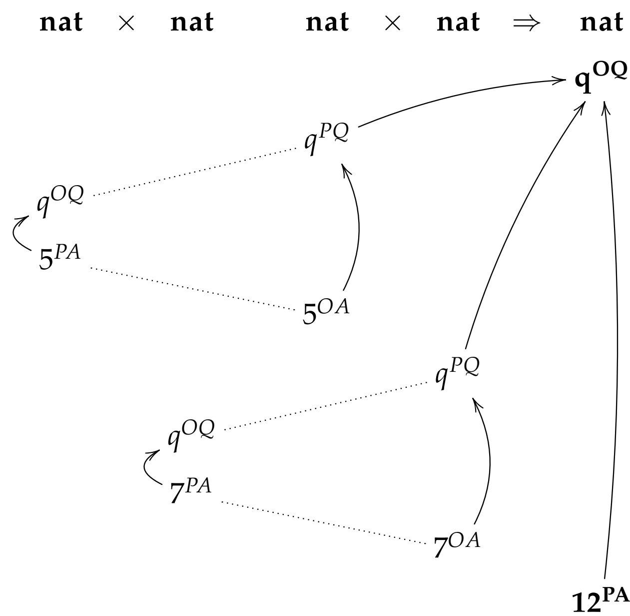


Figure 2.10: The addition of 5 and 7 produced by composition of the appropriate strategies. The dotted lines depict synchronisation while only the bold moves remain after hiding.

Such strategies are called *copycat*, and represent identities for composition.

2.2.2 Concurrency in Game Semantics

The three seminal game semantic models of PCF were strictly alternating: Opponent and Proponent take turns playing. Hence, the first generation of subsequent game semantic models focused on sequential languages, for example, ML and Algol. Introducing concurrency to Game Semantics was a natural next step.

In [AM99c], Abramsky and Melliès describe *concurrent games*, essentially a reformulation of Game Semantics in the tradition of event structures. In this setting, games are defined as complete lattices of plays and strategies as closure operators on them. Concurrent games were then used to model multiplicative-additive linear logic. By contrast, Ghica and Murawski [GM08] used an interleaved form of pointer games to model Idealized Concurrent Algol (ICA). Using a similar approach, Laird gave a game semantics for Idealized CSP [Lai01b] and later to the asynchronous π-calculus [Lai05b]. Finally, Melliès and Mimram combined the two approaches (interleaved and non-interleaved) in their work on *asynchronous games* [MM07]. Melliès later demonstrated how the simply typed lambda calculus can be modelled in this setting [Mel06].

We will briefly discuss, by contrast with earlier models, how concurrency was introduced in Ghica and Murawski's game semantics of ICA.

ICA extends Idealized Algol [Rey81] with static fine-grained shared-variable concurrency. Besides parallel composition, the syntax includes semaphores as a base type for which synchronisation constructs are provided. Like [HO00], the game semantic model uses justified plays—essentially sequences equipped with pointer structures. However, it abandons alternation in favour of interleaved plays that obey the following two rules.

- **Fork:** only a pending question (one that has yet to be answered) can enable further moves.
- **Join:** a question can only be answered once all questions enabled by it have been answered.

Asynchrony is simulated using saturation under certain swappings, a common feature of asynchronous models [Udd86, JJH90]. The salient idea is to close strategies under inessential differences—caused by the asynchronous communication—in the order in which moves are observed.

This is formalised using a preorder $\lesssim$, which we describe next. Let P_A be the set of all justified plays satisfying Fork and Join for arena A. We define a relation $\lesssim_0$ on P_A satisfying, for all $s, s' \in P_A$, we have $s' \lesssim_0 s$ whenever,

1. $s' = s_0 \cdot o \cdot s_1 \cdot s_2$ and $s = s_0 \cdot s_1 \cdot o \cdot s_2$, or
2. $s' = s_0 \cdot s_1 \cdot p \cdot s_2$ and $s = s_0 \cdot p \cdot s_1 \cdot s_2$,

where o is an Opponent move and p is a Proponent move and the justification pointers in s' are 'inherited' from s. We then define the preorder $\lesssim$ on P_A as the least reflexive and transitive relation containing $\lesssim_0$.

Note that since s and s' are legal by definition, the causality structure encoded by justification pointers is sound in s'; in other words, a cause and effect cannot be swapped.

A strategy is called saturated if it is closed under the above permutations; that is, if it is downward closed with respect to $\lesssim$. The identity strategy is obtained by saturating the strictly

alternating copycat strategy. Arenas and saturated strategies are shown to form a Cartesian closed category, and hence, they can be used to model the lambda calculus fragment of the language in a canonical way. Finally, the imperative part of the language is modelled by given morphisms in the category.

2.3 Hardware Compilation via Geometry of Synthesis

Synthesising digital circuits from behavioural specifications written in high-level programming languages, known as *hardware compilation*, is an old idea. However, it has remained a principally academic endeavour for a long time. This can be partly attributed to its inefficacy: high-level specifications can hardly aspire to be as efficient as low-level designs. One of the main reasons for this is poor support for concurrency in general-purpose programming languages. This so-called *semantic gap* [SSC01] focused efforts on extending conventional programming languages with low-level constructs leading to the emergence of languages that are *closer to metal*. Examples include HandelC [Cel], HardwareC [KD90] and Transmogrifier C [Gal95]. These typically extend programming languages (C in particular), or a subset thereof, with lower-level constructs that allow the programmer to use explicit parallelism and introduce temporal constraints.

The advent of reconfigurable hardware and programmable components—for example, Field-Programmable Gate Arrays (FPGAs)—renewed practical interest in hardware compilation. As integrated circuit transistor counts are ever increasing [PPE$^+$97], weaker performance is becoming an acceptable trade-off for lower design costs.

In [Pag96], Page demonstrated that hardware compilation is achievable if performance is sidestepped. Programs are directly translated to a netlist graph (i.e. *boxes and wires*), where variables are mapped into clocked registers and expressions into combinational circuits. The transformation also maps control structures in the programming language into specified control structures in hardware. As a result, compiling imperative programs into hardware can be automated.

Nevertheless, since every expression in the program yields a combinational circuit and every addition symbol results in an adder, the complexity of the output can rapidly become

intractable. The solution, therefore, is to introduce an abstraction mechanism and to handle sharing of circuitry. The common abstraction unit in hardware languages, the *module*, allows to abstract some behaviour and provide it with an interface in such a way that it can be reused. However, sharing these modules is not supported by the hardware languages as this burden is relegated to the programmer.

Building on Page's hardware compilation ideas, Wirth [Wir98] suggests the introduction of subroutines by adding a mechanism for suspending and resuming circuitry and a stack identifying suspended circuits. Each time a subroutine is called, the caller is stopped and only resumed after the called circuit has terminated. Wirth then contends that,

> *"subroutines [. . .] introduce a significant degree of complexity into a circuit. It is indeed highly questionable whether it is worthwhile considering their implementation in hardware."*

Naturally, if subroutines are viewed as an extension or an afterthought, the results may be cumbersome. The Geometry of Synthesis (GoS) hardware compilation framework [Ghi07] takes a more organic approach to the problem. It endows a higher-order imperative programming language with a game-like semantics expressed in terms of circuits. This semantics can subsequently be synthesised into digital circuits. Consequently, higher-order functions are first-class citizens and their bookkeeping is handled through canonical structures borrowed from category theory—in particular, closed monoidal categories. The issue of sharing of resources is tackled by combining an elegant type system due to Reynolds [Rey78] with diagonal morphisms, which Ghica re-brands as *activation managers* in the context of hardware compilation [Ghi09b].

2.3.1 Introduction to GoS

With such notions as process, event and handshake communication, process calculi exhibit the right level of abstraction to act as an interface for hardware compilation. Indeed, there is an important body of work revolving around this idea, for example, [Mar86, Mar87, NvRS88, vS88, OBL98]. Under this light, Geometry of Synthesis (GoS) [Ghi07] can be viewed as a refinement

of this early work by using Game Semantics, in lieu of process calculi, as an intermediary for hardware compilation from programming languages to digital circuits.

The highly intensional nature of game semantics yields a natural correspondence between its fundamental notions and those of digital circuits. Geometry of Synthesis identifies and exploits this fact. If we consider strategies as circuits (or description thereof) and their arenas as circuit interfaces, we get the following analogy.

- An Opponent/Proponent move is an input/output port.
- A question/answer is a request/acknowledgement port.
- A move occurrence is a signal on a port.
- A play is a waveform on an interface.

This idea underlies the formulation of a denotational semantics based on a class of handshake circuits [BKR+91], which can subsequently be refined and synthesised into digital circuits. We begin with an overview of the programming language used in GoS.

2.3.2 Basic Syntactic Control of Interference

Syntactic Control of Interference (SCI) is a typing discipline invented by Reynolds to simplify reasoning about imperative programs, by restricting the way functions interact with their arguments [Rey78]. Its salient feature is ruling out the phenomenon of *covert interference*: when distinct terms can affect each other's outcome in a way that is not syntactically evident. However, interest in SCI went beyond its original stated reason, as it raised several challenging technical issues regarding its type system and semantic model [YH98, OPTT99, Lai05a, McC07].

The programming language resulting from the application of SCI typing to Idealized Algol [Rey81] has been dubbed Basic SCI (BSCI) by O'Hearn [O'H03]. It forbids *aliasing*; so, no distinct identifiers can refer to the same memory location. This is achieved through the use of a *multiplicative* application rule that forbids functions from sharing identifiers with their arguments.

Reddy formulated the first semantic model for BSCI using 'object spaces' [Red96]. His model was later shown to be fully abstract by McCusker [McC02].

The semantic properties of BSCI make it an interesting programming language for particular applications. On the one hand, BSCI is an expressive higher-order imperative ('Algol-like') programming language and its syntactic restrictions rarely impinge on implementing useful algorithms. Moreover, any term of BSCI (with finite data types) can be given a finite-state model [GMO06]. This makes it possible to automatically verify BSCI programs using conventional finite-state model checking techniques [GM06]. For the same reason, it is the language of choice for hardware compilation through Geometry of Synthesis.

The ground types of BSCI are commands, expressions and variables (memory cells), given by the grammar,

$$\sigma ::= \mathsf{com} \mid \mathsf{exp} \mid \mathsf{var}$$

In GoS, expressions and variables are assumed to be boolean in nature. However, this is not assumed in our presentation. BSCI allows product and function types.

$$\theta ::= \sigma \mid \theta \times \theta' \mid \theta \multimap \theta'$$

Typing judgements for terms have the form,

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : A,$$

where x_i are distinct identifiers, A_i and A are types and M is a term. We use $\Gamma, \Delta, \ldots$ to range over *contexts*, i.e., the (unordered) list of identifier type assignments above. Well-typed terms are captured by the following rules.

$$\frac{}{x : A \vdash x : A}\ \text{Axiom}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \multimap B}\ \text{Abstraction}$$

$$\frac{\Delta \vdash M : A \multimap B \qquad \Gamma \vdash N : A}{\Gamma, \Delta \vdash MN : B}\ \text{Application}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}\ \text{Product} \qquad \frac{\Gamma \vdash M : B}{\Gamma, x : A \vdash M : B}\ \text{Weakening}$$

Observe that the application rule above requires Γ and Δ to be disjoint.

The imperative part of BSCI is given by the following constants.

$\mathsf{skip} : \mathsf{com}$	no-op
$n : \mathsf{exp}$	natural number constants
$\circledast : \mathsf{exp} \times \mathsf{exp} \multimap \mathsf{exp}$	arithmetic and arithmetic-logic operators
$; : \mathsf{com} \times A \multimap A$	sequential composition, $A \in \{\mathsf{com}, \mathsf{nat}, \mathsf{var}\}$
$\| : \mathsf{com} \multimap \mathsf{com} \multimap \mathsf{com}$	parallel command composition
$:= : \mathsf{var} \times \mathsf{exp} \multimap \mathsf{com}$	assignment
$! : \mathsf{var} \multimap \mathsf{exp}$	dereferencing
$\mathsf{if} : \mathsf{exp} \times A \times A \multimap A$	selection, $A \in \{\mathsf{com}, \mathsf{exp}, \mathsf{var}\}$
$\mathsf{while} : \mathsf{exp} \times \mathsf{com} \multimap \mathsf{com}$	iteration
$\mathsf{newvar} : (\mathsf{var} \multimap A) \multimap A$	local variable, $A \in \{\mathsf{com}, \mathsf{exp}\}$.

While typing rules allow sharing of identifiers in pairs of terms, they disallow sharing of identifiers between a function and its arguments. It follows that programs using nested function application $(\cdots f(\cdots f(\cdots)\cdots)\cdots)$—in particular, general recursion—do not type. Sequential operators such as arithmetic, composition, and assignment can share arguments and conventional imperative programs, including iterative ones, type correctly. Non-sequential operators ($||$) have a type which prevents sharing of identifiers, and hence race conditions, through the type system; note that this also makes it impossible to implement shared-memory concurrency.

As a final note on expressiveness, SCI can be generalised to a richer type system called *Syntactic Control of Concurrency* (SCC), which allows shared-memory concurrency [GMO06]. Moreover, it was recently shown that almost any recursion-free Idealized Concurrent Algol programs [GM08], barring pathological examples, can be automatically 'serialised' into BSCI via SCC [GS11].

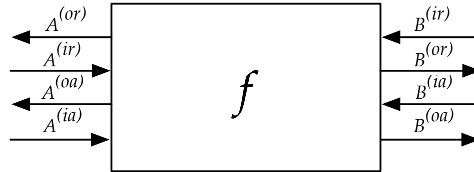
The operational semantics of BSCI is the standard one for an Algol-like language, i.e., call-by-name beta reduction with local-state variable manipulation. A full description may be found in [Ghi07, McC02, McC10].

2.3.3 A Category of Handshake Circuits

In this section and the next one, we review Ghica's denotational semantics for BSCI [Ghi07], expressed in terms of *handshake* circuits. As previously indicated, the adopted abstraction stipulates that circuits are defined by an interface and a behaviour. The interface consists of ports, each of which has a pair of polarities: input (i) or output (o), and request (r) or acknowledgement (a).

The behaviour of a circuit is given in terms of outputs it produces in reaction to inputs from its environment. Input ports can be connected to output ports by extending a wire between them. As a result, a signal is conducted after a *nonzero* delay. Note that the lengths of wires are inconsequential at this level of abstraction.

We will draw these circuits in the same direction as the morphisms they represent. The input/output polarity of each port is further stressed using arrows. For example the circuit corresponding to $f : A \to B$ is



The semantic model is based on a Cartesian category (an affine category with product) of *simple handshake* circuits, which are handshake circuits with constrained behaviour. We begin by outlining the structure of the more general category **HC** of handshake circuits.

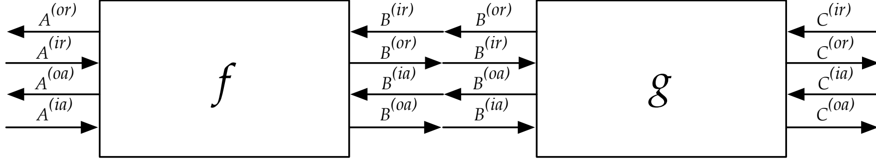
An object A is a set of ports L_A associated with a *polarity function* $\lambda_A : L_A \to \{i, o\} \times \{r, a\}$. We write λ_A^{io} and λ_A^{ra} for the composition of λ_A with the first and second projections respectively. Morphisms $f : A \to B$ are circuits with the following interface.

$$L_{A \to B} = L_A + L_B$$

$$\lambda_{A \to B} = [\langle \lambda_A^{oi}, \lambda_A^{ra} \rangle, \lambda_B]$$

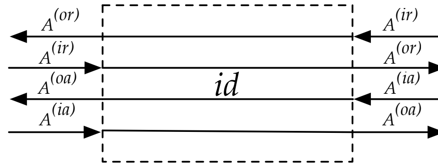
Note that the input/output polarity of the A ports is reversed while their request/acknowledgement polarity remains unchanged.

Given circuits $f : A \to B$ and $g : B \to C$, their composition is the circuit $f;g : A \to C$ formed by connecting f and g as follows.



In the wake of composition, the labels in the B interface become internal. This is reflected in the type of the resulting circuit.

The identity circuit $id : A \to A$ directly connects inputs and outputs with the same request/acknowledgement polarity.

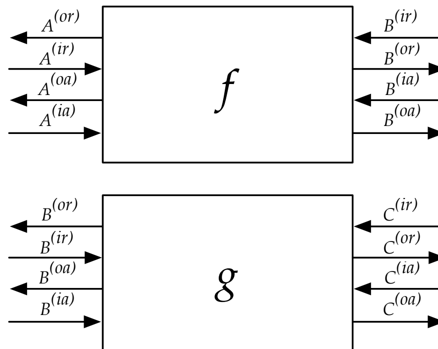


Handshake circuits form a closed monoidal category, whose structure we outline next. The tensor product is defined as the disjoint union on objects,

$$L_{A\otimes B} = L_A + L_B$$

$$\lambda_{A\otimes B} = [\lambda_A, \lambda_B].$$

On morphisms, $f \otimes g : A \otimes C \to B \otimes D$ has the following form.



The unit of the tensor, denoted by I, is the empty set of ports.

Given a pair of objects A and B there is an object $A \Rightarrow B$ defining the interface of a morphism $f : A \to B$ as described above. The evaluation circuit $eval_{A,B} : A_1 \otimes (A_2 \Rightarrow B_1) \to B_2$ is isomorphic to $id_A \otimes id_B$.

Proposition 2.1 ([Ghi07]). **HC** *is a closed symmetric monoidal category.*

In order to allow for circuit reuse, a notion of Cartesian product is necessary. Since, the category **HC** is too degenerate to admit one, the behaviour of its circuits needs to be restricted. These well-behaved circuits, dubbed *simple-handshake*, satisfy the following conditions.

1. **Alternation:** inputs and outputs alternate.
2. **Well-bracketedness:** requests and acknowledgements must be well-bracketed in the game semantic sense [HO00].
3. **Initialisation:** the outermost request is an initial port—one that is drawn from a designated subset of input requests I satisfying, $I_{A \otimes B} = I_{A \times B} = I_A + I_B$ and $I_{A \Rightarrow B} = I_B$.
4. **Seriality:** throughout the lifetime of a circuit, there must not be more than a single action on a request port without an intervening action on its corresponding acknowledgement port.

Proposition 2.2 ([Ghi07]). *The category of simple-handshake circuits* **SHC** *is a Cartesian subcategory of* **HC**.

In the following section, we describe a model of BSCI within the category **SHC**.

2.3.4 A Game-Like Semantics for BSCI

BSCI types are modelled as objects in the category **SHC**. The base types are interpreted as follows.

$$[\![\mathsf{com}]\!] = \{R \mapsto (i,r), D \mapsto (o,a)\}$$

$$[\![\mathsf{exp}]\!] = \{Q \mapsto (i,r), T \mapsto (o,a), F \mapsto (o,a)\}$$

$$[\![\mathsf{var}]\!] = \{WT \mapsto (i,r), WF \mapsto (i,r), Q \mapsto (i,r), D \mapsto (o,a), T \mapsto (o,a), F \mapsto (o,a)\}$$

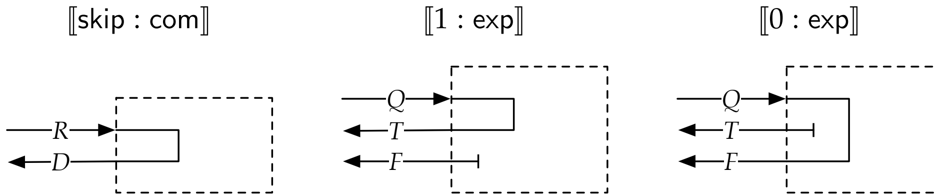
where R stands for 'run', D for 'done' and W for 'write'; and $I_{[\![\mathsf{com}]\!]} = \{R\}$, $I_{[\![\mathsf{exp}]\!]} = \{Q\}$, $I_{[\![\mathsf{var}]\!]} = \{WT, WF, Q\}$. For product and function types we have,

$$[\![\theta \times \theta']\!] = [\![\theta]\!] \times [\![\theta']\!]$$
$$[\![\theta \Rightarrow \theta']\!] = [\![\theta]\!] \Rightarrow [\![\theta']\!].$$

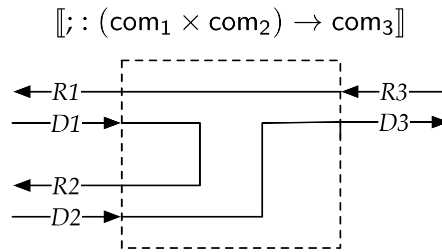
Terms $x_1 : A_1, \ldots, x_n : A_n \vdash M : A$ will be interpreted as a map,

$$\bigotimes_{1 \le i \le n} [\![A_i]\!]_s \xrightarrow{[\![x_1:A_1,\ldots,x_n:A_n \vdash M:A]\!]} [\![A]\!]_s.$$

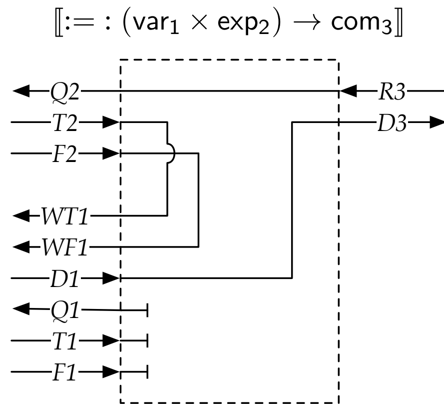
The imperative constants of the language are interpreted using morphisms in **SHC**. We will review these in succession.



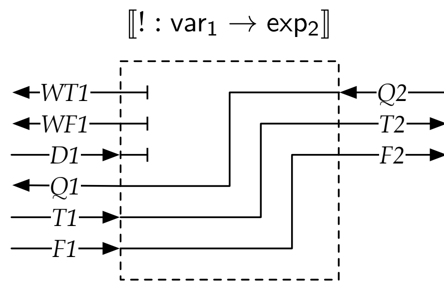
The basic constants are interpreted by immediately responding to an input request on a corresponding output acknowledgement port.



Upon receiving an initial request (R3), the sequential composition circuit sends a run request to its first argument (R1), then, waits until it terminates (D1) to run its second argument (R2). Finally, the circuit acknowledges completion (D3) when its second argument terminates (D2).

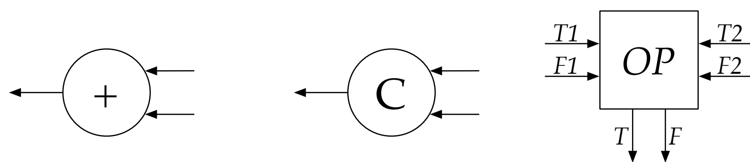


The initial request (R3) on the assignment circuit causes the expression to be evaluated (Q2). Then, depending on the value received (T2/F2), 1 or 0 are written to the first argument (WT1/WF1). The acknowledgement received from the last action (D1) leads the circuit to acknowledge completion (D3).

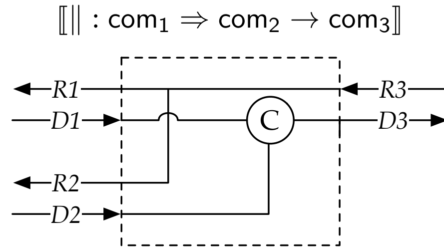


Dereferencing is an even simpler circuit which connects requests and acknowledgements between its argument and its return expression.

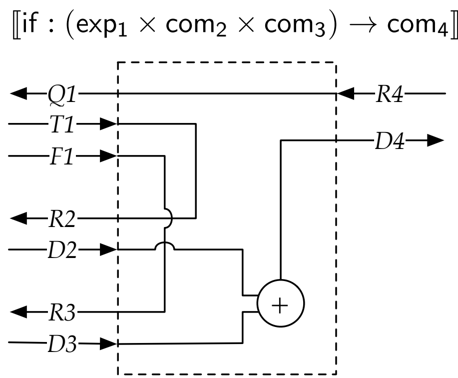
While the circuits so far only required simple wiring, we need to introduce the following auxiliary circuits for more complex control structures.



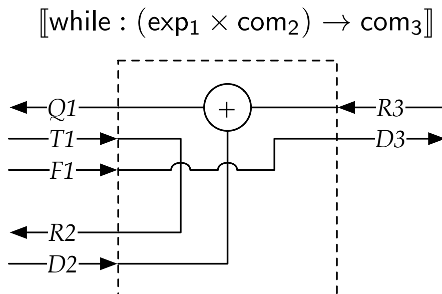
The JOIN circuit (denoted by +) relays any input it receives on either of its two input ports as an output. By contrast, the C-circuit only produces output after receiving signals on both its input ports. Finally, the OP circuit is a stateful, self-resetting circuit that performs logical operations. Note that besides the JOIN circuit, these circuits are not simple handshake.



Parallel composition does not wait for the termination of one of its arguments to execute the other. It terminates when both its arguments do. Naturally, sequential composition can be viewed as straightjacketed version of this circuit. Note that parallel composition is not a simple-handshake circuit because it violates alternation, well-bracketedness and seriality. We further remark that race conditions are prevented by the type system which disallows the sharing of identifiers between concurrent subterms.

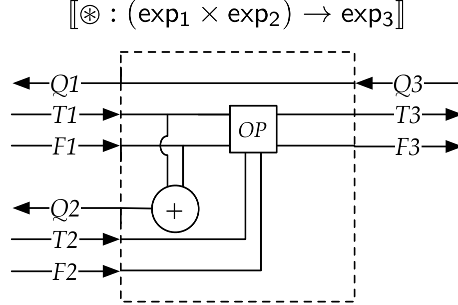


If the guard evaluates to true (T1), the branching circuit initiates its second argument. Otherwise, the third argument is executed. The circuit acknowledges completion whenever the triggered command terminates. Note that well-bracketedness and seriality prevent D2 from acknowledging R3 and D3 from responding to R2.



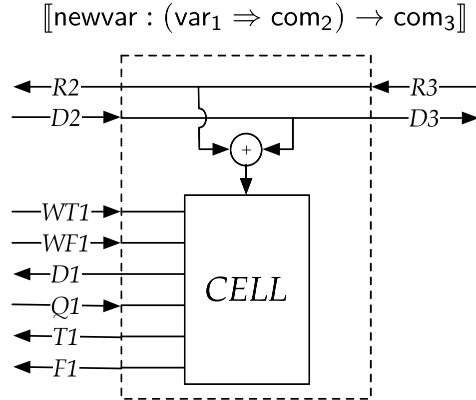
As long as the guard evaluates to true (T1) in the iteration circuit, the body of the loop is executed (R2). The circuit terminates (D3) when the guard becomes false (F1).

$[\![\circledast : (\mathsf{exp}_1 \times \mathsf{exp}_2) \to \mathsf{exp}_3]\!]$



For logical operators, an initial request (Q3) results in the evaluation of the first and second arguments in succession. After that, the result is computed using the OP circuit and returned.

$[\![\mathsf{newvar} : (\mathsf{var}_1 \Rightarrow \mathsf{com}_2) \to \mathsf{com}_3]\!]$



The CELL circuit is a binary memory location. It has type var and therefore behaves accordingly. The second argument is necessary to allow the circuit to be self-resetting. This property, which also holds for all circuits above ensures that circuits behave correctly with the diagonal morphism.

The lambda calculus part of the language is interpreted in the standard way using the structure of the category **SHC**.

$$[\![x : \theta \vdash x : \theta]\!] = id_{[\![\theta]\!]}$$
$$[\![\Gamma, x : \theta \vdash x : \theta]\!] = \,! \otimes id_{[\![\theta]\!]}$$
$$[\![\Gamma \vdash \lambda x.M : \theta' \to \theta]\!] = \Lambda([\![\Gamma, x : \theta' \vdash M : \theta]\!])$$

$$[\![\Gamma, \Delta \vdash FM : \theta]\!] = ([\![\Delta \vdash M : \theta']\!] \otimes [\![\Gamma \vdash F : \theta' \to \theta]\!]); eval$$

$$[\![\Gamma \vdash \langle M, N \rangle : \theta \times \theta']\!] = \delta_{[\![\Gamma]\!]}; ([\![\Gamma \vdash M : \theta]\!] \otimes [\![\Gamma \vdash N : \theta']\!]);$$

A soundness theorem is given, which is also a correctness result for the compilation technique.

Theorem 2.3 ([Ghi07]). *For any closed BSCI term M : com such that $M \Downarrow$, it follows that $[\![M]\!]$ is equivalent to $[\![\mathsf{skip}]\!]$.*

2.4 The Problem in Essence

The purpose of this chapter has been to gather prerequisites for later chapters and situate this thesis in the context of its influences. It also allows us to give a more elaborated description of the problem at hand.

The game semantic models found in the literature, like their process calculus forerunners, are asynchronous: moves cannot be observed *simultaneously*. For example, the identity in Game Semantics is the *copycat strategy*, where one player mimics the actions of the other. However, this strategy proceeds in steps: one player issues a move *then* the other repeats it. Hence, the copycat strategy fits within an asynchronous model of communication commonly found in hardware where connectors have an arbitrary delay. By contrast, connectors in clocked synchronous platforms conduct input to output within the same clock cycle. In abstract terms, this represents a *synchronous wire* that instantly copies any input it receives on one side to the other.

Such synchronous identities have previously appeared in synchronous models, for example, in the interaction category **SProc** [AGN96]. Hence, reconciling the asynchronous nature of the copycat strategy with synchronous behaviour is achievable. Nevertheless, doing so further raises challenges in composition as phenomena akin to *instant feedback*, *causal loops* and *schizophrenia* appear.

In practical terms, Geometry of Synthesis [Ghi07] allows asynchronous games to be directly compiled into asynchronous circuits [GS10]. This is characterised by the restriction that wires

conduct signals after a bounded nonzero delay. However, given that synchronous platforms (e.g. FPGAs) are now pervasive, it is also desirable to be able to compile into synchronous clocked circuits. The discrepancy between the asynchronous input and synchronous output of the compiler is resolved in an *ad hoc* fashion. The compiler introduces additional delays (in the form of flip-flops), which negatively impacts the efficiency of the resulting circuitry.

An optimisation can be subsequently applied to produce more efficient circuits with lower latency. Initially, we developed an algorithm based on *round abstraction* [AH99]—especially in its formulation over automata [AHR98]—to fulfil this task. It operates on BSCI strategies constructs represented as transition systems. In essence, the idea is to group as many transitions as *legally* possible onto a single one. The algorithm forbids the aggregation of transitions that have the same label; as this corresponds, in the circuitry, to receiving the same signal several times on the same port, within the same clock cycle. The algorithm improves the efficiency of the resulting circuitry, since widely used circuits in the hardware compiler, such as the identity, no longer introduce any delays [Ghi09b].

One of the main motivations of our work stems from this dichotomy between the input and output of the GoS hardware compiler: respectively, asynchronous games and synchronous digital circuits. Optimisation post compilation, as described above, does not guarantee correctness. In fact, it may produce erroneous circuits. For example, contrary to initial intuitions suggesting that simultaneous read and write accesses should be allowed, we see in Chapter 4 that such behaviour may lead to race conditions.

Ultimately, we aim to develop a game semantics that appropriately captures the synchronous concurrency encountered at the level of hardware. This will not only subsume the aforementioned optimisation, but also result in provably correct circuits that compose well.

The present thesis is a first step towards that goal. It lays the foundations for a new research programme focusing on round abstraction as a technique relating asynchronous and synchronous frameworks *compositionally*. We are especially interested in synchronous and asynchronous models that have a game-like structure. This leads us to study round abstraction on non-causal processes, then on process augmented with justification pointers, and finally, on interleaved asynchronous processes. We also explore how synchronous processes can be extended with global clocks and how the notion of determinism is affected by moving from

sequential to synchronous traces.

2.4.1 Methodology

We aim at reusing existing game semantic models by applying the technique of round abstraction [AH99] to them. This first requires defining round abstraction within a compositional setting and establishing its basic properties, a focal point in this thesis. By keeping our study sufficiently general, our results are not limited to this application, and should transpose without difficulty to other low-level concurrent models.

In the course of this thesis, we define several categories. We view the construction of each category as a sanity check. Each studied phenomenon—for example, causality, global clocks and determinism—requires definitions that usually stem from corresponding physical processes. We construct categories to check that the definitions are sensible and capture the correct intuition.

Chapter 3

A Trace Model of Processes*

We introduce the trace model that will underlie our study of round abstraction. Since we focus on compositionality, it is formulated in the style of game models for concurrency [GMO06, GM08]. We introduce the ability to describe synchronous behaviour by updating the classical notion of trace as a sequence [Hoa85] to allow for more than a single event to be observed at the same time. By analogy, we have the following correspondence between the concepts introduced in this chapter and those from Game Semantics.

signature	arena
trace	position / play
process	strategy
label	move
event	move occurrence

In Section 3.1, we introduce *locally synchronous traces* as a means to describe synchronous behaviours. We then lift these to synchronous processes and find that they form a category. Later, in Section 3.2, we detail the categorical structure of the model.

3.1 Traces and Processes

We construct a low-level model of concurrency where events are atomic: not implicitly buffered or tagged. As a result, events are consumed as they are produced. Processes describe system behaviour over time. Each process has an interface, which we call its *signature*, and its behaviour is described by a set of traces of input and output events over the signature. Each trace

*Extended and revised version of [GM10, Sec. 2]

corresponds to a possible behaviour record or history. This vision accords with what Francez *et al.* [FHLR79] call *a priori* semantics: the behaviour of a process is given by the set of all its possible behaviours when placed in any environment.

3.1.1 Signatures

Definition 3.1 (Signature)**.** *A signature A is a finite set equipped with a labelling function. Formally, it is a pair $\langle L_A, \pi_A \rangle$ where,*

- L_A *is a finite* set of labels*.*
- $\pi_A : L_A \to \{i, o\}$ *maps each label to an* input/output *polarity.*

Signatures are akin to game semantic *arenas*—more accurately, pre-arenas [Wal04]. We call the elements of L_A, depending on context, *labels* or *ports*. The input and output polarities are not absolute: they refer to how a port is perceived by the process. We will see that, in composition, the same port will be an input to one process and an output to another.

We write π^* for a labelling function that is like π, but has the input/output polarities reversed; similarly for arena A^*. Note that this operation is an involution; that is, $(\pi^*)^* = \pi$.

We introduce two composite signatures: a *tensor* signature ($\otimes$) and an *arrow* signature ($\Rightarrow$). Intuitively, the former amounts to grouping two signatures together, while the latter additionally reverses the polarities of the left operand. They are defined as follows.

$$\begin{aligned} L_{A\otimes B} &= L_A + L_B \\ \pi_{A\otimes B} &= [\pi_A, \pi_B] \end{aligned}$$

$$\begin{aligned} L_{A\Rightarrow B} &= L_A + L_B \\ \pi_{A\Rightarrow B} &= [\pi_A^*, \pi_B] \end{aligned}$$

Note the similarity of these definitions to those of the *product* and *exponential* arenas in Game Semantics [HO00, AJM00]. Furthermore, note that polarity may be ignored in this chapter. The proofs and definitions, tensor and arrow signatures notwithstanding, remain the same. This is not the case for the refined trace model in Chapter 5.

3.1.2 Traces

We update the classical notion of trace—typically given as a string, sequence or totally-ordered set. To allow for more than a single event to be observed simultaneously, we instead use a preordered set to describe the temporal position of each event. Additionally, we disentangle a common ambiguity between a label and its occurrence in a trace. For example, such ambiguity can be found in Mazurkiewicz traces [Maz95] and Game Semantics [GM08], where a may correspond to either the event a or the sequence consisting of the single event a. We instead use a carrier set together with a labelling function. We begin with the general notion of *pre-traces*.

Definition 3.2 (Locally synchronous pre-trace). *A* locally synchronous pre-trace s *over a signature* A *is a triple* $\langle E_s, \preceq_s, \lambda_s \rangle$ *where* E_s *is a finite set of events,* $\preceq_s$ *is a total preorder on* E_s *and* $\lambda_s : E_s \to L_A$ *is a function mapping events to labels in* A.

The total preorder signifies temporal precedence; for an element $e \in E_s$, if $\lambda_s(e) = a \in L_A$ we say that e is an *occurrence* of a, or an a-event. We denote by $\Delta(A)$ the set of pre-traces over A. It is convenient to define the following notion.

Definition 3.3 (Simultaneity). *Given a pre-trace* $\langle E, \preceq, \lambda \rangle$ *over signature* A*, we say that two events* $e_1, e_2 \in E$ *are* simultaneous, *written* $e_1 \approx e_2$ *if* $e_1 \preceq e_2$ *and* $e_2 \preceq e_1$.

In each pre-trace, the total preorder $\preceq$ induces a total order on the equivalence classes of the associated equivalence relation $\approx$. We interpret each equivalence class as a collection of 'simultaneous' events.

Example 3.4. For illustration, we will often use a simplified notation that reflects the intuition of Remark 3.8. So, the pre-trace,

$$\langle \{e_1, e_2, e_3, e_4\}, tr(\{(e_1, e_2), (e_2, e_3), (e_3, e_4), (e_4, e_3)\}), \lambda \rangle$$

over A where $L_A = \{a, b, c\}$, tr denotes transitive closure and $\lambda = \{e_1 \mapsto a, e_2 \mapsto b, e_3 \mapsto a, e_4 \mapsto c\}$ is simply denoted by $a.b.\langle a, c \rangle$. The pre-trace consists of an a-event, followed by a b-event, followed by an a-event and a c-event at the same time.

We will work with pre-traces that respect the following restriction.

Definition 3.5 (Singularity)**.** *The events of a pre-trace $\langle E, \preceq, \lambda \rangle$ over signature A are* singular *if for any two events $e_1, e_2 \in E$, if $e_1 \approx e_2$ and $\lambda(e_1) = \lambda(e_2)$, then $e_1 = e_2$.*

A pre-trace has singular events if it does not have any distinct simultaneous occurrences of the same label (or, intuitively, if it represents a sequence of sets according to the notation of Example 3.4). This restriction is not inherent to synchronous concurrency but, is essential for modelling low-level behaviour where events are atomic, i.e., not implicitly buffered or tagged. By definition, we rule out phenomena akin to *schizophrenia* in Esterel [BG88, SW01]. We call a pre-trace satisfying singularity, a trace.

Definition 3.6 (Locally synchronous trace)**.** *A* locally synchronous trace *is a pre-trace with singular events.*

We denote by $\Theta(A)$ the set of traces over A. Traces are equivalent if there is a bijection between their carrier sets acting homomorphically on event labelling and temporal ordering.

Definition 3.7 (Trace equivalence)**.** *Two pre-traces are considered equivalent, written $s \cong t$, if they only differ in the choice of their carrier sets. Formally, pre-traces $s = \langle E_s, \preceq_s, \lambda_s \rangle$ and $t = \langle E_t, \preceq_t, \lambda_t \rangle$ are equivalent if there exists a bijection $\phi : E_s \to E_t$ satisfying $(\forall e, e' \in E_s)(e \preceq_s e' \Leftrightarrow \phi(e) \preceq_t \phi(e'))$ and $\lambda_s = \lambda_t \circ \phi$.*

In practice, since the choice of carrier sets is irrelevant, we will work with the quotient sets $\Delta/\cong$ and $\Theta/\cong$, only distinguishing pre-traces and traces up to $\cong$-equivalence. In the sequel, for traces s and t, we will write $s = t$ to mean that they belong to the same equivalence class.

Remark 3.8. The definition of pre-traces may look, at first, unnecessarily low-level and convoluted. A more direct characterisation is to see a pre-trace as a sequence of nonempty multisets. However, while this alternative view is simpler and closer to intuition, it makes subsequent definitions more complex. Our definition has the additional advantage of offering a straightforward way to compare pre-traces for synchrony: a pre-trace s is more synchronous than t if, roughly, $\preceq_t \subseteq \preceq_s$. This will turn out to be important as we will see in Chapter 4.

Our conception of synchrony is a minimalistic one; time is discretised and events can be simultaneous, which is the salient feature of a synchronous process [BG88]. However, our notion of trace does not rely on a global clock. Instead, we assume that each system has its own

internal and abstract clock, relative to which simultaneity is defined; and that these clocks can compose. The notion of synchrony we have is a local one [Cha84].

In Chapter 6, we will see that the local synchrony assumption is sufficiently expressive. We also see that our setting can be lifted, in a principled way, to a globally synchronous one.

We define the *concatenation* of two traces at the level of rounds; that is, all the events of the second trace come after the events of the first one.

Definition 3.9 (Trace concatenation)**.** *The* concatenation *of two traces* $s = \langle E_s, \preceq_s, \lambda_s \rangle$, $t = \langle E_t, \preceq_t, \lambda_t \rangle$*, denoted by* $s \cdot t$*, is the trace defined by the triple* $\langle E_s + E_t, \preceq_s + \preceq_t + (E_s \times E_t), [\lambda_s, \lambda_t] \rangle$.

Trace fusion is like concatenation but the final round of the first trace and the initial round of the second trace are taken to be simultaneous. Let the last round of a trace s be defined in the obvious way, $last(s) = \{e \in E_s \mid (\forall e' \in E_s)(e' \preceq_s e)\}$. The *first* round is defined in an analogous way.

Definition 3.10 (Trace fusion)**.** *The* fusion *of two traces* s*,* t*, denoted by* $s * t$*, is the trace defined by the quadruple* $\langle E_s + E_t, \preceq', [\lambda_s, \lambda_t] \rangle$*, where* $\preceq' = \preceq_s + \preceq_t + E_s \times E_t + first(t) \times last(s)$.

3.1.3 Processes

Using the definitions of traces, we can now introduce processes. We define a *prefix* preorder $\preccurlyeq$ on $\Theta(A)$ as the least reflexive and transitive relation containing $\cong$ and satisfying for traces s and t, $s \preccurlyeq s \cdot t$. A set S is said to be downward closed with respect to an order, say $\preccurlyeq$, if $y \preccurlyeq x$ and $x \in S$ implies $y \in S$.

Definition 3.11 (Process)**.** *A* process σ *over signature* A*,* $\sigma : A$*, is a nonempty and* $\preccurlyeq$*-downward closed set of traces.*

Processes that are $\preccurlyeq$-closed are said to be *prefix-closed*. Given an arbitrary set of traces τ, let $pc(\tau)$ be the smallest process that contains τ.

For any signatures A, B and C we define the set of *interaction traces*, written $int(A, B, C)$, as $\Theta((A \Rightarrow B) \Rightarrow C)$. Given another signature D, let $int(A, B, C, D)$ be the set $\Theta(((A \Rightarrow B) \Rightarrow C) \Rightarrow D)$. Let $s \upharpoonright A$ be the trace obtained from s by deleting all events with labels not belonging to L_A.

Composition of processes is defined similarly to game semantic composition. It can be understood as parallel composition followed by hiding, in the process calculi sense.

Definition 3.12 (Interaction). *Let $\sigma : A \to B$ and $\tau : B \to C$ be two processes. Their interaction is $\sigma \between \tau = \{u \in int(A, B, C) \mid u \upharpoonright A + B \in \sigma$ and $u \upharpoonright B + C \in \tau\}$.*

Definition 3.13 (Composition). *Let $\sigma : A \to B$ and $\tau : B \to C$ be two processes. Their composition is $\sigma;\tau : A \to C = \{u \upharpoonright A + C \mid u \in \sigma \between \tau\}$.*

The result below indicates that the formalism so far makes sense, allowing us to model the function space and function application.

Theorem 3.14. *Processes form a closed symmetric monoidal category called* **SynProc**.

3.2 Categorical Structure

This section presents the details of the categorical structure of our trace model, and may be skipped without loss of continuity.

The category **SynProc** of processes has

- signatures $A, B, C \ldots$ as objects,
- processes $\sigma, \tau, \upsilon \ldots$ as morphisms,
- a composition operation on pairs of processes,

$$\frac{\sigma : A \to B \quad \tau : B \to C}{\sigma;\tau : A \to C}$$

- for any object A, an identity morphism $id_A : A \to A$ defined below.

The identity morphism is an instantaneous version of the *copycat strategy* in Game Semantics [GM08].

Definition 3.15 (Identity morphism). *The identity morphism for object A, denoted by id_A, is the process consisting of traces u over $A \Rightarrow A$ satisfying, for all events e in E_u, there exists an event $e' \neq e$, such that $e \approx_u e'$ and $[\lambda_u(e) = inl(a)$ if and only if $\lambda_u(e') = inr(a)]$ and $[\lambda_u(e) = inr(a)$ if and only if $\lambda_u(e') = inl(a)]$, where $a \in L_A$.*

We will need the following definitions.

Definition 3.16 (Trace injection). *Injection on traces is a function* $in : \Theta(A) \to \Theta(A+B)$, *such that for every* $t = \langle E, \preceq, \lambda \rangle$ *in* $\Theta(A)$

$$in(t) = \langle E, \preceq, \lambda' \rangle$$

where $\lambda' = lin \circ \lambda$ *and* lin *is the injection on labels* $lin : L_A \to L_A + L_B$.

In this thesis, we sometimes abuse notation by omitting label injections when no confusion may arise.

Definition 3.17 (Trace projection). *Projection on traces is a function* $out : \Theta(A+B) \to \Theta(A)$ *such that for every* $t = \langle E, \preceq, \lambda \rangle$ *in* $\Theta(A+B)$

$$out(t) = \langle E', \preceq', \lambda' \rangle$$

where

- $E' = \{e \in E \mid \lambda(e) \in lin(L_A)\}$,
- $\preceq' = \preceq \cap (E' \times E')$,
- $\lambda' = lout \circ \lambda$ *where* $lout$ *is the projection on labels* $lout : L_A + L_B \rightharpoonup L_A$.

Projection on labels is a partial function since it is undefined for labels in L_B. We alternatively denote trace projection using $\restriction$. For example, we may write the projection above as $t \restriction A$. If u is a trace over $B+C$, we may write $u \restriction^B_C$ to emphasise that the projection deletes C-events and keeps B-events. We denote by $\sigma \restriction A$ the projection over A applied pointwise on a process σ.

We begin by showing that **SynProc** is a category. The next two lemmas demonstrate that singularity is preserved by the basic operations on traces.

Lemma 3.18. *Injection on traces preserves singularity.*

Proof. Let $s = \langle E, \preceq, \lambda \rangle$ be a trace over $A+B$. Let $in(s) = \langle E, \preceq, lin \circ \lambda \rangle$, where $lin : L_A \to L_A + L_B$, be an injection. If $e, e' \in E$ and $e \approx e'$ and $(lin \circ \lambda)(e) = (lin \circ \lambda)(e')$, then $\lambda(e) = \lambda(e')$ because lin is injective. It follows that $e = e'$ because s is a trace. ∎

Lemma 3.19. *Projection preserves the temporal order, i.e. if s is a trace over $A+B$ and $e, e' \in E_s$ such that $\lambda_s(e), \lambda_s(e') \in lin(L_A)$ and $e \preceq_s e'$, then $e \preceq_{s \upharpoonright A} e'$.*

Proof. This follows from the definition of projection as, $\preceq_{s \upharpoonright A} = \preceq_s \cap (E' \times E')$ where $E' = \{e \in E_s \mid \lambda_s(e) \in lin(L_A)\}$. ■

Lemma 3.20. *Projection on traces preserves singularity.*

Proof. Let s be a trace over $A+B$. Let $s \upharpoonright A = \langle E_0 = \{e \in E_s \mid \lambda_s(e) \in lin(L_A)\}, \preceq_0 = \preceq_s \cap (E_0 \times E_0), lout \circ \lambda_s \rangle$, where $lout : L_A + L_B \rightharpoonup L_A$, be the projection of s over A. Let us denote by $\approx_0$ the simultaneity relation of the projection; that is, $\approx_0 = \preceq_0 \cap \preceq_0^{-1}$. If $e, e' \in E_0$ and $e \approx_0 e'$ and $(lout \circ \lambda_s)(e) = (lout \circ \lambda_s)(e')$, then $\lambda_s(e) = \lambda_s(e')$ because $lout$ is a partial injection and is defined on $\lambda_s(e), \lambda_s(e')$. Additionally, we have $e \approx_s e'$ since projection preserves the temporal order by Lemma 3.19. Finally, we conclude that $e = e'$ because s is a trace. ■

In the following lemmas, we show that composition is well defined.

Lemma 3.21. *Composition preserves singularity.*

Proof. Let $\sigma : A \to B$ and $\tau : B \to C$ be processes. Following Lemma 3.20, it is sufficient to show that interaction preserves singularity. Let $u \in \sigma \between \tau$. Suppose $e, e' \in E_u$ and $e \approx_u e'$ and $\lambda_u(e) = \lambda_u(e')$. We have the following cases.

- If $\lambda_u(e), \lambda_u(e') \in L_A$, then $e \approx_{u \upharpoonright A+B} e'$ (because projection preserves the temporal order by Lemma 3.19) and $\lambda_{u \upharpoonright A+B}(e) = \lambda_{u \upharpoonright A+B}(e')$. Since $u \upharpoonright A+B \in \sigma$ and σ is a process, we deduce that $e = e'$.

- If $\lambda_u(e), \lambda_u(e') \in L_C$, then $e \approx_{u \upharpoonright B+C} e'$ (because projection preserves the temporal order by Lemma 3.19) and $\lambda_{u \upharpoonright B+C}(e) = \lambda_{u \upharpoonright B+C}(e')$. Since $u \upharpoonright B+C \in \tau$ and τ is a process, we deduce that $e = e'$.

- If $\lambda_u(e), \lambda_u(e') \in L_B$, we use either previous argument to deduce that $e = e'$. ■

We will need the following lemma in the sequel.

Lemma 3.22. *Projection distributes over concatenation. That is, for any trace $u = s \cdot t \in \Theta(A + B)$, $u \upharpoonright A = (s \upharpoonright A) \cdot (t \upharpoonright A)$.*

Proof. This follows from the definitions of trace concatenation (Definition 3.9) and trace projection (Definition 3.17). ■

The next lemma shows that composition preserves the basic property of a process.

Lemma 3.23. *Composition preserves prefix-closure.*

Proof. Let $\sigma : A \to B$ and $\tau : B \to C$ be processes. We begin by showing that $\sigma \between \tau$ is prefix-closed. Take $u \cdot u' \in \sigma \between \tau$. By Definition 3.12, this entails $u \cdot u' \upharpoonright A + B \in \sigma$ and $u \cdot u' \upharpoonright B + C \in \tau$. Using Lemma 3.22, we get $u \cdot u' \upharpoonright A + B = (u \upharpoonright A + B) \cdot (u' \upharpoonright A + B)$ and $u \cdot u' \upharpoonright B + C = (u \upharpoonright B + C) \cdot (u' \upharpoonright B + C)$. So, $(u \upharpoonright A + B) \cdot (u' \upharpoonright A + B) \in \sigma$ and $(u \upharpoonright B + C) \cdot (u' \upharpoonright B + C) \in \tau$. Since σ and τ are prefix-closed, we get $u \upharpoonright A + B \in \sigma$ and $u \upharpoonright B + C \in \tau$. This implies, by Definition 3.12, that $u \in \sigma \between \tau$. So, $\sigma \between \tau$ is prefix-closed.

Next, we show that $\sigma ; \tau$ is prefix-closed. Let $v = v_1 \cdot v_2 \in \sigma ; \tau$. By Definition 3.13, we know that there exists $u \in \sigma \between \tau$ such that $u \upharpoonright A + C = v$. Without loss of generality, let $E_v = E_{u \upharpoonright A+C}$.

We show that $u = u_1 \cdot u_2$ and $u_1 \upharpoonright A + C = v_1$. Let u_1 be defined as follows.

- $E_{u_1} = E_{v_1} + \{e \in E_u \mid \lambda_u(e) \in L_B \text{ and } (\forall e' \in last(v_1))(e \preceq_u e')\}$
- $\preceq_{u_1} = \preceq_u \cap (E_{u_1} \times E_{u_1})$
- $\lambda_{u_1} = \lambda_u \upharpoonright E_{u_1}$

Let u_2 be defined as follows.

- $E_{u_2} = E_{v_2} + \{e \in E_u \mid \lambda_u(e) \in L_B \text{ and } (\forall e' \in last(v_1))(e' \preceq_u e \text{ and } e \not\preceq_u e')\}$
- $\preceq_{u_2} = \preceq_u \cap (E_{u_2} \times E_{u_2})$
- $\lambda_{u_2} = \lambda_u \upharpoonright E_{u_2}$

Note that the decomposition of u into u_1 and u_2 is unique—i.e. $E_{u_1} \cap E_{u_2} = \emptyset$ and $E_{u_1} \cup E_{u_2} = E_u$. It is also clear, by construction, that $u_1 \upharpoonright A + C = v_1$. Since $\sigma \between \tau$ is prefix-closed, $u_1 \in \sigma \between \tau$ and hence $v \in \sigma ; \tau$. ■

Now, we can prove that the basic axioms of a category hold.

Lemma 3.24. *For each object A in* **SynProc***, id_A is the identity morphism.*

Proof. Let $\sigma : A \to B$ be a process and $id_B : B \to B'$. We want to show that $\sigma; id_B = \sigma$.

We begin by proving that $\sigma; id_B \subseteq \sigma$. After unfolding the definitions, this is equivalent to proving that for any $u \in int(A, B, B')$

$$\text{if } u \restriction A + B = s \in \sigma \tag{3.1}$$

$$\text{and } u \restriction B + B' \in id_B \tag{3.2}$$

$$\text{then } u \restriction A + B' = s \tag{3.3}$$

Let (3.1) and (3.2) be true. To prove (3.3), we need to show a bijection $\phi : E_s \to E_{u \restriction A+B'}$ such that $(\forall e_1, e_2 \in E_s)(e_1 \preceq_s e_2 \text{ iff } \phi(e_1) \preceq_{u \restriction A+B'} \phi(e_2))$ and $\lambda_s = \lambda_{u \restriction A+B'} \circ \phi$. We define ϕ as follows.

$$\phi(e) = \begin{cases} e & \text{if } \lambda_s(e) \in inl(L_A) \\ e' & \text{if } \lambda_s(e) \in inr(L_B), \text{ where } e' \approx_u e \text{ and } \lambda_u(e) = inl(inr(b)) \text{ and } \lambda_u(e') = inr(b) \\ & \text{and } b \in L_B \end{cases}$$

We know ϕ is well-defined because the definition of *id* ensures the existence and singularity ensures the uniqueness of the output. In addition, its inverse ϕ^{-1} is defined as,

$$\phi^{-1}(e) = \begin{cases} e & \text{if } \lambda_{u \restriction A+B'}(e) \in inl(L_A) \\ e' & \text{if } \lambda_{u \restriction A+B'}(e) \in inr(L_{B'}), \text{ where } e' \approx_u e \text{ and } \lambda_u(e) = inr(b) \\ & \text{and } \lambda_u(e') = inl(inr(b)) \text{ and } b \in L_B \end{cases}$$

and is a function. Therefore, ϕ is a bijection. Next, we show that $e_1 \preceq_s e_2$ iff $\phi(e_1) \preceq_{u \restriction A+B'} \phi(e_2)$. First, we prove the left to right implication through a case analysis. For all $e, e' \in E_s$,

- if $e \preceq_s e'$ and $\phi(e) = e$ and $\phi(e') = e'$, then $\phi(e) \preceq_{u \restriction A+B'} \phi(e')$
- if $e \preceq_s e'$ and $\phi(e) = e$ and $\phi(e') \approx_u e'$, then $\phi(e) \preceq_{u \restriction A+B'} \phi(e')$
- if $e \preceq_s e'$ and $\phi(e) \approx_u e$ and $\phi(e') = e'$, then $\phi(e) \preceq_{u \restriction A+B'} \phi(e')$

- if $e \preceq_s e'$ and $\phi(e) \approx_u e$ and $\phi(e') \approx_u e'$, then $\phi(e) \preceq_{u\restriction A+B'} \phi(e')$

For the right to left implication, we use the definition of ϕ^{-1}. For all $e, e' \in E_{u\restriction A+B'}$,

- if $e \preceq_{u\restriction A+B'} e'$, then and $\phi^{-1}(e) = e$ and $\phi^{-1}(e') = e'$, then $\phi^{-1}(e) \preceq_s \phi^{-1}(e')$
- if $e \preceq_{u\restriction A+B'} e'$, then and $\phi^{-1}(e) = e$ and $\phi^{-1}(e') \approx_u e'$, then $\phi^{-1}(e) \preceq_s \phi^{-1}(e')$
- if $e \preceq_{u\restriction A+B'} e'$, then and $\phi^{-1}(e) \approx_u e$ and $\phi^{-1}(e') = e'$, then $\phi^{-1}(e) \preceq_s \phi^{-1}(e')$
- if $e \preceq_{u\restriction A+B'} e'$, then and $\phi^{-1}(e) \approx_u e$ and $\phi^{-1}(e') \approx_u e'$, then $\phi^{-1}(e) \preceq_s \phi^{-1}(e')$

Finally, we show that $\lambda_s = \lambda_{u\restriction A+B'} \circ \phi$. First, note that domains of the two functions are ϕ-bijective and the codomains are equal ($L_A + L_B = L_A + L_{B'}$). We show that for all $e \in E_s$, we have $\lambda_s(e) = \lambda_{u\restriction A+B'} \circ \phi(e)$.

Let $\lambda_s(e) = inl(a), a \in L_A$. It follows, by definition of ϕ, that $\phi(e) = e$. We also have if $\lambda_s(e) = inl(a), a \in L_A$, then $\lambda_u(e) = inl(inl(a))$, which in turn implies that $\lambda_{u\restriction A+B'}(e) = inl(a)$. So, $\lambda_{u\restriction A+B'} \circ \phi(e) = inl(a) = \lambda_s(e)$. Let $\lambda_s(e) = inr(b), b \in L_B$. It follows, by definition of ϕ, that $\phi(e) = e'$, where $e \approx_u e'$ and $\lambda_u(e) = inl(inr(b))$ and $\lambda_u(e') = inr(b)$. So, $\lambda_{u\restriction A+B'} \circ \phi(e) = inr(b) = \lambda_s(e)$.

Next, we prove $\sigma \subseteq \sigma; id_B$. Let $s \in \sigma$. We will find an interaction trace $u \in int(A, B, B')$ such that $u \restriction A + B = s$ and $u \restriction B + B' \in id_B$ and $u \restriction A + B' = s$. Let u be defined as follows.

- $E_u = E_{s\restriction A} + E_{s\restriction B} + E_{s\restriction B}$. Let us refer to these subsets using injections $in_1 : E_{s\restriction A} \to E_u$, $in_2 : E_{s\restriction B} \to E_u$ and $in_3 : E_{s\restriction B} \to E_u$.
- $\lambda_u = \lambda_{s\restriction A} + \lambda_{s\restriction B} + \lambda_{s\restriction B}$
- The preorder $\preceq_u$ is defined as follows. For all $e, e' \in E_s$,

  P1 if $e, e' \in E_{s\restriction A}$, then $e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_1(e')$

  P2 if $e \in E_{s\restriction A}$ and $e' \in E_{s\restriction B}$, then $[e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_1(e) \preceq_u in_3(e')]$

  P3 if $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $[e \preceq_s e' \Leftrightarrow in_2(e) \preceq_u in_1(e') \Leftrightarrow in_3(e) \preceq_u in_1(e')]$

  P4 if $e, e' \in E_{s\restriction B}$, then $[e \preceq_s e' \Leftrightarrow in_2(e) \preceq_u in_2(e') \Leftrightarrow in_3(e) \preceq_u in_3(e') \Leftrightarrow in_2(e) \preceq_u in_3(e') \Leftrightarrow in_3(e) \preceq_u in_2(e')]$

We first argue that $\preceq_u$ is a total preorder. As $\preceq_s$ is reflexive, (P1) and (P4) ensure that $\preceq_u$ is reflexive too. Since $\preceq_s$ is total, $\preceq_u$ is also total because every two events are comparable: A-events are comparable with A-events by (P1), A-events are comparable with B-events by (P2) and (P3), A-events are comparable with B'-events by (P2) and (P3), B-events are comparable with B-events by (P4), B-events are comparable with B'-events by (P4) and B'-events are comparable with B'-events by (P4). Next, we show that $\preceq_u$ is transitive. The proof can be achieved via a long case analysis (27 cases). We will consider all cases at once by using subscript variables. Let $in_j(e_1) \preceq_u in_k(e_2)$ and $in_k(e_2) \preceq_u in_l(e_3)$ where $j,k,l \in \{1,2,3\}$. By definition of $\preceq_u$, we have $e_1 \preceq_s e_2$ and $e_2 \preceq_s e_3$. Because $\preceq_s$ is transitive, we get $e_1 \preceq_s e_3$. By inspecting the definition of $\preceq_u$, we get $in_j(e_1) \preceq_u in_l(e_3)$.

Note that u has singular events because s respects singularity and the B'-events that are made simultaneous with B-events in (P4) are labelled by a different copy of B (i.e. B').

Finally, we show that u satisfies the conditions set out above. Because $\preceq_s$ is reflexive, (P4) implies that $(\forall e \in E_{s\restriction B})(in_2(e) \approx_u in_3(e))$. We also have $\lambda_{u\restriction B+B'}(in_2(e)) = inl(b)$ if and only if $\lambda_{u\restriction B+B'}(in_3(e)) = inr(b), b \in L_B$. So, $u \restriction B + B' \in id_B$. We briefly argue that $u \restriction A + B = s$. We have $E_{u\restriction A+B} = E_{s\restriction A} + E_{s\restriction B}$. There is a bijection $\phi : E_s \to E_{u\restriction A+B}$ defined as

$$\phi(e) = \begin{cases} in_1(e) & \text{if } \lambda_s(e) \in inl(L_A) \\ in_2(e) & \text{if } \lambda_s(e) \in inr(L_B) \end{cases}$$

We need to prove that $(\forall e, e' \in E_s)(e \preceq_s e' \text{ iff } \phi(e) \preceq_{u\restriction A+B} \phi(e'))$. We have $e \preceq_s e' \Leftrightarrow in_j(e) \preceq_u in_k(e') \Leftrightarrow in_j(e) \preceq_{u\restriction A+B} in_k(e')$ where $j,k \in \{1,2\}$.

The proof that $u \restriction A + B' = s$ is similar.

The proof that $id_A; \sigma = \sigma$ is similar to the proof of $\sigma; id_B = \sigma$. ■

The following ancillary concept will be useful in the sequel.

Definition 3.25 (Trace composition). *The interaction of traces $s \in \Theta(A \Rightarrow B)$ and $t \in \Theta(B \Rightarrow C)$ is given by $s \lightning t = \{u \in int(A,B,C) \mid u \restriction A + B = s \text{ and } u \restriction B + C = t\}$ and their composition by $s; t = \{u \restriction A + C \mid u \in s \lightning t\}$.*

Lemma 3.26. *Let s be an interaction trace over $int(A,B,C)$ and t be a trace over $C \to D$. We have $(s \upharpoonright_B^{A+C}) \lightning t = (s \lightning t) \upharpoonright_B^{A+C+D}$.*

Proof. First, we prove $(s \upharpoonright_B^{A+C}) \lightning t \subseteq (s \lightning t) \upharpoonright_B^{A+C+D}$. Let $u \in (s \upharpoonright_B^{A+C}) \lightning t$; that is, $u \in int(A,C,D)$ and $u \upharpoonright A+C = s \upharpoonright A+C$ and $u \upharpoonright C+D = t$. To show that $u \in (s \lightning t) \upharpoonright_B^{A+C+D}$, we construct a trace $v \in int(A,B,C,D)$ such that $v \upharpoonright A+C+D = u$ and $v \upharpoonright A+B+C = s$ and $v \upharpoonright C+D = t$ as follows. Without loss of generality, we assume that $E_{s \upharpoonright A+C} = E_{u \upharpoonright A+C}$. This allows us to simplify the proof by avoiding the use of trace equivalence isomorphisms.

- $E_v = E_{u\upharpoonright A} + E_{s\upharpoonright B} + E_{u\upharpoonright C} + E_{u\upharpoonright D}$. In the sequel, we will omit injections on events; for example, we will write e where it should be $in_j(e), j \in \{1,2,3,4\}$. This is possible because the sets $E_{u\upharpoonright A}, E_{s\upharpoonright B}, E_{u\upharpoonright C}$ and $E_{u\upharpoonright D}$ are disjoint. It should improve readability at the cost of abusing notation.

- $\lambda_v = \lambda_{u\upharpoonright A} + \lambda_{s\upharpoonright B} + \lambda_{u\upharpoonright C} + \lambda_{u\upharpoonright D}$

- We define $\preceq_v$ in two steps. Let $\preceq_1$ be the relation satisfying the following.

  – $(\forall e, e' \in E_{u\upharpoonright A} + E_{u\upharpoonright C} + E_{u\upharpoonright D})(e \preceq_u e' \Leftrightarrow e \preceq_1 e')$

  – $(\forall e, e' \in E_{u\upharpoonright A} + E_{s\upharpoonright B} + E_{u\upharpoonright C})(e \preceq_s e' \Leftrightarrow e \preceq_1 e')$

  Note that the only events that are not comparable using $\preceq_1$ are B-events with respect to D-events. We define a second relation $\preceq_2$ as follow. For all $e \in E_{s\upharpoonright B}$ and $e' \in E_{u\upharpoonright D}$

  – $(\exists e'' \in E_v)(e' \preceq_1 e''$ and $e'' \preceq_1 e)$ if and only if $e' \preceq_2 e$

  – $(\exists e'' \in E_v)(e \preceq_1 e''$ and $e'' \preceq_1 e')$ or $(\forall e'' \in E_v)\big((e \not\preceq_1 e''$ or $e'' \not\preceq_1 e')$ and $(e' \not\preceq_1 e''$ or $e'' \not\preceq_1 e)\big)$ if and only if $e \preceq_2 e'$

  We then set $\preceq_v$ to be the union of $\preceq_1$ and $\preceq_2$.

We first show that $\preceq_v$ is a total preorder. Since $\preceq_u$ and $\preceq_s$ are reflexive, we know that $\preceq_1$ is reflexive. So, $\preceq_v$ is reflexive. Totality of $\preceq_v$ is ensured by $\preceq_2$. Next, we show that $\preceq_v$ is transitive, i.e. for all $e, e', e'' \in E_v$, if $e \preceq_v e'$ and $e' \preceq_v e''$, then $e \preceq_v e''$. Let $e \preceq_v e'$ and $e' \preceq_v e''$. The proof can be achieved via a case analysis of 64 cases. Forty eight of these are covered by the definition of $\preceq_1$ and the fact that $\preceq_u$ and $\preceq_s$ are transitive. The remaining 16 cases are those

where one event in $\{e, e', e''\}$ is a B-event and another is a D-event. Discarding symmetric cases, we consider the following three nontrivial cases.

1. If $e \in E_{s\restriction B}$ and [$e' \in E_{u\restriction A}$ or $e' \in E_{u\restriction C}$] and $e'' \in E_{u\restriction D}$, we use the definition of $\preceq_2$ to conclude that $e \preceq_v e''$.

2. If $e \in E_{u\restriction A}$ and $e' \in E_{s\restriction B}$ and and $e'' \in E_{u\restriction D}$, then by the definition of $\preceq_2$ we have the following two cases.

   (a) There is $e_0 \in E_v$ such that $e' \preceq_1 e_0$ and $e_0 \preceq_1 e''$. Note that e_0 is either an A-event or a C-event because it is related to both e' and e''. We have $e \preceq_v e' \Leftrightarrow e \preceq_s e'$ and $e' \preceq_1 e_0 \Leftrightarrow e' \preceq_s e_0$. Since $\preceq_s$ is transitive, we get $e \preceq_s e_0$. Since $u \restriction A + C = s \restriction A + C$, we get $e \preceq_u e_0$. We also have $e_0 \preceq_1 e'' \Leftrightarrow e_0 \preceq_u e''$. Using the transitivity of u, we conclude that $e \preceq_u e''$. We then use the definition of $\preceq_1$ to conclude that $e \preceq_v e''$.

   (b) There is no $e_0 \in E_v$ satisfying $e' \preceq_1 e_0 \preceq_1 e''$ or $e'' \preceq_1 e_0 \preceq_1 e'$. Let us refer to this assumption by ($\star$). Since $\preceq_u$ is total, we have $e \preceq_u e''$ or $e'' \preceq_u e$. Assume to the contrary that $e'' \preceq_u e$. By definition of $\preceq_v$, this is equivalent to $e'' \preceq_1 e$. Since $e \preceq_v e' \Leftrightarrow e \preceq_s e' \Leftrightarrow e \preceq_1 e'$, it contradicts ($\star$). So, $e \preceq_u e''$ and therefore $e \preceq_v e''$.

3. If $e \in E_{u\restriction B}$ and $e' \in E_{s\restriction D}$ and and $e'' \in E_{u\restriction B}$, then by the definition of $\preceq_2$ we have the following two cases.

   (a) There are $e_0, e_2 \in E_v$ such that $e \preceq_1 e_0$ and $e_0 \preceq_1 e'$ and $e' \preceq_1 e_2$ and $e_2 \preceq_1 e''$. Note that e_0, e_2 are labelled by A or C because they are related to both e' and e''. We have $e_0 \preceq_u e_2$ by the transitivity of $\preceq_u$. It follows $e_0 \preceq_s e_2$ since $u \restriction A + C = s \restriction A + C$. Since $\preceq_s$ is transitive, we get $e_0 \preceq_s e''$. We use the transitivity of $\preceq_s$ again to conclude that $e \preceq_s e''$ and therefore $e \preceq_v e''$.

   (b) There is no $e_0 \in E_v$ satisfying $e \preceq_1 e_0 \preceq_1 e'$ or $e' \preceq_1 e_0 \preceq_1 e$ and there is $e_2 \in E_v$ such that $e' \preceq_1 e_2$ and $e_2 \preceq_1 e''$. Let us refer to this assumption by ($\bigstar$). Since $\preceq_s$ is total, we have $e \preceq_s e''$ or $e'' \preceq_s e$. Assume to the contrary that $e'' \preceq_s e$. By definition of $\preceq_v$, this is equivalent to $e'' \preceq_1 e$. We also have $e_2 \preceq_1 e'' \Leftrightarrow e_2 \preceq_s e''$. Since $\preceq_s$ is transitive, we get $e_2 \preceq_s e \Leftrightarrow e_2 \preceq_1 e$. However, we already have $e' \preceq_1 e_2$ so $e' \preceq_1 e_2$ and $e_2 \preceq_1 e$ which contradicts ($\bigstar$). So, $e \preceq_s e''$ and therefore $e \preceq_v e''$.

The interaction trace v respects singularity because s has singular events and u has singular events and $s \upharpoonright A + C = u \upharpoonright A + C$; and $\preceq_2$ only relates B-event and D-events. Now we show that the conditions set above are satisfied by $\preceq_v$.

We first prove that $v \upharpoonright A + B + C = s$. We have $E_{v \upharpoonright A+B+C} = E_{s \upharpoonright A} + E_{s \upharpoonright B} + E_{s \upharpoonright C}$. There is a bijection $\phi : E_s \to E_{v \upharpoonright A+B+C}$ defined as

$$\phi(e) = \begin{cases} in_1(e) & \text{if } \lambda_s(e) \in inl(inl(L_A)) \\ in_2(e) & \text{if } \lambda_s(e) \in inl(inr(L_B)) \\ in_3(e) & \text{if } \lambda_s(e) \in inr(L_C) \end{cases}$$

We need to prove that for all $e, e' \in E_s$, we have $e \preceq_s e'$ iff $\phi(e) \preceq_{v \upharpoonright A+B+C} \phi(e')$. We have $e \preceq_s e' \Leftrightarrow in_j(e) \preceq_1 in_k(e') \Leftrightarrow in_j(e) \preceq_v in_k(e') \Leftrightarrow in_j(e) \preceq_{v \upharpoonright A+B+C} in_k(e')$ where $j, k \in \{1, 2, 3\}$.

Next, we show that $v \upharpoonright A + C + D = u$. We have $E_{v \upharpoonright A+C+D} = E_{u \upharpoonright A} + E_{u \upharpoonright C} + E_{u \upharpoonright D}$. There is a bijection $\psi : E_u \to E_{v \upharpoonright A+C+D}$ defined as

$$\psi(e) = \begin{cases} in_1(e) & \text{if } \lambda_u(e) \in inl(inl(L_A)) \\ in_2(e) & \text{if } \lambda_u(e) \in inl(inr(L_C)) \\ in_4(e) & \text{if } \lambda_u(e) \in inr(L_D) \end{cases}$$

We need to prove that for all $e, e' \in E_u$, we have $e \preceq_u e'$ iff $\psi(e) \preceq_{v \upharpoonright A+C+D} \psi(e')$. We have $e \preceq_u e' \Leftrightarrow in_j(e) \preceq_1 in_k(e') \Leftrightarrow in_j(e) \preceq_v in_k(e') \Leftrightarrow in_j(e) \preceq_{v \upharpoonright A+C+D} in_k(e')$ where $j, k \in \{1, 2, 3\}$.

Finally, $v \upharpoonright C + D = t$ follows from the following two facts: $v \upharpoonright A + C + D = u$ and $u \upharpoonright C + D = t$.

Now, we prove $(s \upharpoonright_B^{A+C}) \lightning t \supseteq (s \lightning t) \upharpoonright_B^{A+C+D}$. Let $u \in s \lightning t$; that is, $u \in int(A, B, C, D)$ and $u \upharpoonright A + B + C = s$ and $u \upharpoonright C + D = t$. Applying the projection over $A + C + D$, we get $u \upharpoonright A + C + D \in int(A, C, D)$ and $u \upharpoonright A + C = s \upharpoonright A + C$ and $u \upharpoonright C + D = t$. So, $u \in (s \upharpoonright_B^{A+C}) \lightning t$. ■

Corollary 3.27. *Let $\sigma : A \to B$, $\gamma : B \to C$ and $\tau : C \to D$ be processes. We have $(\sigma \lightning \gamma \upharpoonright_B^{A+C}) \lightning \tau = ((\sigma \lightning \gamma) \lightning \tau) \upharpoonright_B^{A+C+D}$.*

Lemma 3.28. *Let s be a trace in $\Theta(A + B + C)$. We have $(s \upharpoonright_C^{A+B}) \upharpoonright_B^A = s \upharpoonright_{B+C}^A$.*

Proof. This is trivial as removing all C-events, then all B-events from s yields the same trace as removing all events labelled by either B or C in one step. ■

Lemma 3.29. *Composition is associative.*

Proof. Let $\sigma : A \to B$, $\tau : B \to C$ and $\gamma : C \to D$ be morphisms in **SynProc**. To prove that composition is associative, i.e. $(\sigma;\tau);\gamma = \sigma;(\tau;\gamma)$, we first show that interaction is associative. That is, $(\sigma \between \tau) \between \gamma = \sigma \between (\tau \between \gamma)$. We expand $(\sigma \between \tau) \between \gamma$ using Definition 3.12,

$$\begin{aligned}(\sigma \between \tau) \between \gamma &= \{u \in int(A,B,C,D) \mid u \restriction_{D}^{A+B+C} \in \sigma \between \tau \text{ and } u \restriction_{A+B}^{C+D} \in \gamma\} \\ &= \{u \in int(A,B,C,D) \mid (u \restriction_{D}^{A+B+C}) \restriction_{C}^{A+B} \in \sigma \text{ and } (u \restriction_{D}^{A+B+C}) \restriction_{A}^{B+C} \in \tau \\ &\qquad \text{and } u \restriction_{A+B}^{C+D} \in \gamma\}\end{aligned}$$

Using Lemma 3.28, we get,

$$(\sigma \between \tau) \between \gamma = \{u \in int(A,B,C,D) \mid u \restriction_{C+D}^{A+B} \in \sigma \text{ and } u \restriction_{A+D}^{B+C} \in \tau \text{ and } u \restriction_{A+B}^{C+D} \in \gamma\}$$

Expanding $\sigma \between (\tau \between \gamma)$ using Definition 3.12 yields,

$$\begin{aligned}\sigma \between (\tau \between \gamma) &= \{u \in int(A,B,C,D) \mid u \restriction_{C+D}^{A+B} \in \sigma \text{ and } u \restriction_{A}^{B+C+D} \in \tau \between \gamma\} \\ &= \{u \in int(A,B,C,D) \mid u \restriction_{C+D}^{A+B} \in \sigma \text{ and } (u \restriction_{A}^{B+C+D}) \restriction_{D}^{B+C} \in \tau \\ &\qquad \text{and } (u \restriction_{A}^{B+C+D}) \restriction_{B}^{C+D} \in \gamma\}\end{aligned}$$

Using Lemma 3.28, we get,

$$\sigma \between (\tau \between \gamma) = \{u \in int(A,B,C,D) \mid u \restriction_{C+D}^{A+B} \in \sigma \text{ and } u \restriction_{A+D}^{B+C} \in \tau \text{ and } u \restriction_{A+B}^{C+D} \in \gamma\}$$

So, $(\sigma \between \tau) \between \gamma = \sigma \between (\tau \between \gamma)$. We denote this result by ($\bigstar$).

In the next step, we use the above intermediate result to show that composition is associative. By Definition 3.13, composition consists in hiding the internal channels using projection

after interaction. So, we can write $\sigma;\tau = \sigma \lightning \tau \upharpoonright_B^{A+C}$. Hence,

$$\begin{aligned}
(\sigma;\tau);\gamma &= ((\sigma \lightning \tau \upharpoonright_B^{A+C}) \lightning \gamma) \upharpoonright_C^{A+D} \\
&= ((\sigma \lightning \tau) \lightning \gamma \upharpoonright_B^{A+C+D}) \upharpoonright_C^{A+D} \text{ by Corollary 3.27} \\
&= (\sigma \lightning \tau) \lightning \gamma \upharpoonright_{B+C}^{A+D} \text{ by Lemma 3.28} \\
&= \sigma \lightning (\tau \lightning \gamma) \upharpoonright_{B+C}^{A+D} \text{ by } (\bigstar) \\
&= (\sigma \lightning (\tau \lightning \gamma) \upharpoonright_C^{A+B+D}) \upharpoonright_B^{A+D} \text{ by Lemma 3.28} \\
&= (\sigma \lightning (\tau \lightning \gamma \upharpoonright_C^{B+D})) \upharpoonright_B^{A+D} \text{ by Corollary 3.27} \\
&= \sigma;(\tau;\gamma).
\end{aligned}$$

■

The following Proposition sums up our results so far.

Proposition 3.30. **SynProc** *is a category.*

3.2.1 Monoidal Structure

The monoidal structure is defined by the identity object I, which is the signature with an empty set of labels $\emptyset$, and a notion of *tensor product*. This structure is subject to the usual coherence conditions. Intuitively, tensoring signatures amounts to taking their disjoint union, while tensored processes are interleaved.

Definition 3.31 (Tensor product)**.** *The tensor product is a bifunctor* $\otimes$: **SynProc** $\times$ **SynProc** $\to$ **SynProc** *defined as follows.*

- *On objects: see the definition of tensor signature in Section 3.1.1.*
- *On morphisms: for processes $\sigma : A \to B$, $\tau : C \to D$, their tensor product is,*

$$\sigma \otimes \tau = \{t \in \Delta(A \otimes C \Rightarrow B \otimes D) \mid t \upharpoonright A + B \in \sigma \textit{ and } t \upharpoonright C + D \in \tau\}$$

Next, we show that **SynProc** satisfies the axioms of a monoidal category.

Lemma 3.32. *The tensor product preserves singularity.*

Proof. Let $\sigma : A \to B$ and $\tau : C \to D$ be processes. Tensoring processes preserves singularity as labels are disjoint in $A \Rightarrow B$ and $C \Rightarrow D$. ■

Lemma 3.33. *The tensor product preserves prefix-closure.*

Proof. We use the same strategy underlying the first part of the proof of Lemma 3.23. ■

Lemma 3.34. *The tensor product preserves identity morphisms; that is, $id_A \otimes id_B = id_{A\otimes B}$.*

Proof. For $u \in id_A$ and events $e, e' \in E_u$, where $u \in id_A$, let us write $e \leftrightharpoons_u e'$ when $e \approx_u e'$ and $[\lambda_u(e) = inl(a)$ if and only if $\lambda_u(e') = inr(a)]$ and $[\lambda_u(e) = inr(a)$ if and only if $\lambda_u(e') = inl(a)]$, where $a \in L_A$. We want to show that $id_A \otimes id_B = id_{A\otimes B}$.

First, we show that $id_A \otimes id_B \subseteq id_{A\otimes B}$. Let $u \in id_A \otimes id_B$; that is, $u \in \Theta(A \otimes B \Rightarrow A' \otimes B')$ and $u \restriction A + A' \in id_A$ and $u \restriction B + B' \in id_B$. Using Definition 3.15, we get $u \in \Theta(A \otimes B \Rightarrow A' \otimes B')$ and $(\forall e \in E_{u\restriction A+A'}, \exists e' \in E_{u\restriction A+A'})(e \leftrightharpoons_{u\restriction A+A'} e')$ and $(\forall e \in E_{u\restriction B+B'}, \exists e' \in E_{u\restriction B+B'})(e \leftrightharpoons_{u\restriction B+B'} e')$. Since $E_u = E_{u\restriction A+A'} \cup E_{u\restriction B+B'}$ and $(\forall e, e' \in E_u)$(if $e \approx_{u\restriction A+A'} e'$ or $e \approx_{u\restriction B+B'} e'$, then $e \approx_u e'$), we conclude that for all $e \in E_u$, there is $e \in E_u$ such that $e \leftrightharpoons_u e'$. Therefore, $u \in id_{A\otimes B}$.

Now, we show that $id_A \otimes id_B \supseteq id_{A\otimes B}$. Let $u \in id_{A\otimes B}$; that is, $u \in \Theta(A \otimes B \Rightarrow A' \otimes B')$ and $(\forall e \in E_u, \exists e \in E_u)(e \leftrightharpoons_u e')$. After projecting u over $A + A'$, we get $u \restriction A + A' \in \Theta(A \Rightarrow A')$ and $(\forall e \in E_{u\restriction A+A'}, \exists e' \in E_{u\restriction A+A'})(e \leftrightharpoons_{u\restriction A+A'} e')$; that is, $u \restriction A + A' \in id_A$. Projecting u over $B + B'$ yields $u \restriction B + B' \in \Theta(B \Rightarrow B')$ and $(\forall e \in E_{u\restriction B+B'}, \exists e' \in E_{u\restriction B+B'})(e \leftrightharpoons_{u\restriction B+B'} e')$; that is, $u \restriction B + B' \in id_B$. So, $u \in id_A \otimes id_B$. ■

Lemma 3.35. *Let $\sigma : A \to B$, $\gamma : B \to C$ and $\tau : D \to E$ be processes. We have $(\sigma \between \gamma \restriction_B^{A+C}) \otimes \tau = ((\sigma \between \gamma) \otimes \tau) \restriction_B^{A+C+D+E}$.*

Proof. We use the same strategy underlying the proof of Lemma 3.26. ■

Lemma 3.36. *The tensor product preserves composition.*

Proof. Let $\sigma : A \to B$, $\tau : B \to C$, $\alpha : D \to E$ and $\beta : E \to F$ be processes. We need to show that $(\sigma; \tau) \otimes (\alpha; \beta) = (\sigma \otimes \alpha); (\tau \otimes \beta)$. We first prove that $(\sigma \between \tau) \otimes (\alpha \between \beta) = (\sigma \otimes \alpha) \between (\tau \otimes \beta)$.

$$LHS = \{u \in \Delta((A \otimes D \Rightarrow B \otimes E) \Rightarrow C \otimes F) \mid u \restriction A{+}B{+}C \in \sigma \between \tau \text{ and } u \restriction D{+}E{+}F \in \alpha \between \beta\}$$

$$= \{u \in \Delta((A \otimes D \Rightarrow B \otimes E) \Rightarrow C \otimes F) \mid u \upharpoonright A{+}B \in \sigma \text{ and } u \upharpoonright B{+}C \in \tau \text{ and } u \upharpoonright D{+}E \in \alpha$$
$$\text{and } u \upharpoonright E + F \in \beta\}$$

$$RHS = \{u \in \Delta((A \otimes D \Rightarrow B \otimes E) \Rightarrow C \otimes F) \mid u \upharpoonright A{+}D{+}B{+}E \in \sigma \otimes \alpha \text{ and}$$
$$u \upharpoonright B{+}E{+}C{+}F \in \tau \otimes \beta\}$$
$$= \{u \in \Delta((A \otimes D \Rightarrow B \otimes E) \Rightarrow C \otimes F) \mid u \upharpoonright A{+}B \in \sigma \text{ and } u \upharpoonright B{+}C \in \tau \text{ and } u \upharpoonright D{+}E \in \alpha$$
$$\text{and } u \upharpoonright E + F \in \beta\}$$

Hence, $LHS = RHS$. Let us refer to this intermediate result by $(\star)$. Next, we show that $(\sigma;\tau) \otimes (\alpha;\beta) = (\sigma \otimes \alpha);(\tau \otimes \beta)$.

$$\begin{aligned}(\sigma;\tau) \otimes (\alpha;\beta) &= (\sigma \between \tau \upharpoonright A + C) \otimes (\alpha \between \beta \upharpoonright D + F) \text{ by Definition 3.13} \\ &= \big((\sigma \between \tau) \otimes (\alpha \between \beta)\big) \upharpoonright A + C + D + F \text{ by Lemma 3.35} \\ &= \big((\sigma \otimes \alpha) \between (\tau \otimes \beta)\big) \upharpoonright A + C + D + F \text{ by } (\star) \\ &= (\sigma \otimes \alpha);(\tau \otimes \beta)\end{aligned}$$

■

Next, we show the existence of the monoidal natural isomorphisms. The proofs of naturality are omitted.

Lemma 3.37. *The tensor product is associative. That is, there exists a natural isomorphism called the* associator*, assigning to each triple of objects A, B, C an isomorphism,*

$$\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$$

such that the pentagon diagram in Figure 3.1 commutes.

Proof. It is clear from the definition of tensor product that the only difference between signatures $(A \otimes B) \otimes C$ and $A \otimes (B \otimes C)$ is in the tagging of labels in the disjoint union. In other words, $(A \otimes B) \otimes C$ and $A \otimes (B \otimes C)$ are equivalent up to associativity of disjoint union. Therefore, there exists an isomorphism $(A \otimes B) \otimes C \to A \otimes (B \otimes C)$ for every triple of objects A, B, C, whose action is to retag the labels and whose traces are equivalent, up to retagging of labels by inl and inr, to those in $id_{A \otimes B \otimes C}$. We call this isomorphism $\alpha_{A,B,C}$. The coherence condition in

Figure 3.1 then follows from the definitions of the tensor product and α. ■

Lemma 3.38. *The tensor product has I as left and right identity. That is, there exists two natural isomorphisms called* left and right unitors, *assigning to each object A the following isomorphisms,*

$$\lambda_A : I \otimes A \to A$$

$$\rho_A : A \otimes I \to A$$

such that the triangle diagram in Figure 3.2 commutes.

Proof. Since I is the object with an empty label set, $\langle \emptyset, \pi_\emptyset \rangle$, it follows that for any $A \in \mathbf{SynProc}$, $I \otimes A$ and $A \otimes I$ are equivalent to A up to tagging of A-events in the disjoint union. Therefore, we define the left identity isomorphism λ_A and the right identity isomorphism ρ_A as those with the signatures above, and which consist of exactly the same traces, up to retagging of labels by *inl* and *inr*, as id_A. The coherence condition in Figure 3.2 directly follows from the definitions of the tensor product, λ, and ρ. ■

In the next part, we show that **SynProc** is symmetric.

Proposition 3.39. SynProc *is a symmetric monoidal category. That is, there exists a natural isomorphism called* symmetry *that assigns to every pair of objects A, B an isomorphism, $\gamma_{A,B} : A \otimes B \to B \otimes A$ such that the hexagon diagrams in Figure 3.3 commute and $\gamma_{A,B}; \gamma_{B,A} = id_{A \otimes B}$.*

Proof. For any $A, B \in \mathbf{SynProc}$, let $\gamma_{A,B} : A \otimes B \to B \otimes A$ be the isomorphism consisting of exactly the same traces, up to retagging of labels by *inl* and *inr*, as $id_{A \otimes B}$. It satisfies the coherence conditions in Figure 3.3. It is then clear that $\gamma_{A,B}; \gamma_{B,A}$ has the same traces, up to retagging of labels by *inl* and *inr*, as $id_{A \otimes B}; id_{A \otimes B}$. However, as $\gamma_{A,B}$ changes the tags of labels and $\gamma_{B,A}$ changes them back according to ($\star$), $\gamma_{A,B}; \gamma_{B,A} = id_{A \otimes B}; id_{A \otimes B} = id_{A \otimes B}$.

$$L_{A \otimes B \Rightarrow A' \otimes B'} = inl(inl(A)) \cup inl(inr(B)) \cup inr(inl(A')) \cup inr(inr(B'))$$

$$L_{A \otimes B \Rightarrow B' \otimes A'} = inl(inl(A)) \cup inl(inr(B)) \cup inr(inl(B')) \cup inr(inr(A')) \quad (\star)$$

$$L_{B' \otimes A' \Rightarrow A \otimes B} = inl(inl(B')) \cup inl(inr(A')) \cup inr(inl(A)) \cup inr(inr(B))$$

■

$$(A \otimes (B \otimes C)) \otimes D$$

$$\alpha_{A,B,C} \otimes id_D \qquad \alpha_{A,B \otimes C,D}$$

$$((A \otimes B) \otimes C) \otimes D \qquad A \otimes ((B \otimes C) \otimes D)$$

$$\alpha_{A \otimes B,C,D} \qquad id_A \otimes \alpha_{B,C,D}$$

$$(A \otimes B) \otimes (C \otimes D) \xrightarrow{\alpha_{A,B,C \otimes D}} A \otimes (B \otimes (C \otimes D))$$

Figure 3.1: Coherence condition for α

$$(A \otimes I) \otimes B \xrightarrow{\alpha_{A,I,B}} A \otimes (I \otimes B)$$

$$\rho_A \otimes id_B \qquad id_A \otimes \lambda_B$$

$$A \otimes B$$

Figure 3.2: Coherence condition for λ and ρ

We can now state the following.

Proposition 3.40. **SynProc** *is a symmetric monoidal category.*

3.2.2 Closed Structure

The closed structure requires every two objects A and B in the category to have an object $A \Rightarrow B$, called the exponential, that may be used to model the function space, and an *evaluation morphism*, $eval_{A,B}$, that may be used to model function application. The former is the arrow signature introduced in Section 3.1.1. The latter is isomorphic to $id_{A \otimes B}$.

Equivalently, **SynProc** is a closed symmetric monoidal category if there exists a natural isomorphism Λ that assigns to any objects A, B, C a bijection $\Lambda_{A,B,C}$: **SynProc**$(A \otimes B, C) \rightarrow$ **SynProc**$(A, B \Rightarrow C)$. We define $\Lambda_{A,B,C}$ as follows. For all $\sigma : A \otimes B \rightarrow C$, $s \in \sigma \Leftrightarrow \psi(s) \in \Lambda_{A,B,C}(\sigma)$ where $\psi(s)$ is defined by the following.

$$
\begin{array}{ccccc}
A \otimes (B \otimes C) & \xrightarrow{\alpha^{-1}_{A,B,C}} & (A \otimes B) \otimes C & \xrightarrow{\gamma_{A,B} \otimes id_C} & (B \otimes A) \otimes C \\
{\scriptstyle \gamma_{A,B\otimes C}} \downarrow & & & & \downarrow {\scriptstyle \alpha_{B,A,C}} \\
(B \otimes C) \otimes A & \xleftarrow[\alpha^{-1}_{B,C,A}]{} & B \otimes (C \otimes A) & \xleftarrow[id_B \otimes \gamma_{A,C}]{} & B \otimes (A \otimes C)
\end{array}
$$

$$
\begin{array}{ccccc}
(A \otimes B) \otimes C & \xrightarrow{\alpha_{A,B,C}} & A \otimes (B \otimes C) & \xrightarrow{id_A \otimes \gamma_{B,C}} & A \otimes (C \otimes B) \\
{\scriptstyle \gamma_{A\otimes B,C}} \downarrow & & & & \downarrow {\scriptstyle \alpha^{-1}_{A,C,B}} \\
C \otimes (A \otimes B) & \xleftarrow[\alpha_{C,A,B}]{} & (C \otimes A) \otimes B & \xleftarrow[\gamma_{A,C} \otimes id_B]{} & (A \otimes C) \otimes B
\end{array}
$$

Figure 3.3: Coherence conditions for γ

- $E_{\psi(s)} = E_s$
- $\preceq_{\psi(s)} = \preceq_s$
- $\lambda_{\psi(s)}$ is defined as follows.

$$
\lambda_{\psi(s)}(e) = \begin{cases} inl(a) & \text{if } \lambda_s(e) = inl(inl(a)), a \in L_A \\ inr(inl(b)) & \text{if } \lambda_s(e) = inl(inr(b)), b \in L_B \\ inr(inr(c)) & \text{if } \lambda_s(e) = inr(c), c \in L_C \end{cases}
$$

Definition 3.41 (Evaluation morphism). *The evaluation morphism* $eval_{A,B} : (A \Rightarrow B) \otimes A \to B$ *is given by,* $eval_{A,B} = \Lambda^{-1}_{A\Rightarrow B,A,B}(id_{A\Rightarrow B}) = \{t \in \Theta(((A_1 \Rightarrow B_1) \otimes A_2) \Rightarrow B_2) \mid t \restriction A_1 + A_2 \in id_A$ *and* $t \restriction B_1 + B_2 \in id_B\}$.

We can now show that **SynProc** is closed.

Proposition 3.42. **SynProc** *with* $- \otimes -$, I, $eval_{A,B}$ *is a closed symmetric monoidal category.*

Proof. We show that for every morphism $\sigma : A \otimes B \to C$, we have $(\Lambda_{A,B,C}(\sigma) \otimes id_B); eval_{B,C} = \sigma$. Let $id_B : B' \to B''$ and $eval_{B,C} : B'' \otimes (B \Rightarrow C) \to C'$.

We begin by proving that $(\Lambda_{A,B,C}(\sigma) \otimes id_B); eval_{B,C} \subseteq \sigma$. After unfolding the definitions,

this is equivalent to proving that for any $u \in int(A \otimes B', (B \Rightarrow C) \otimes B'', C')$

$$\text{if } u \upharpoonright A + B + C = t \in \Lambda_{A,B,C}(\sigma) \tag{3.4}$$

$$\text{and } u \upharpoonright B' + B'' \in id_B \tag{3.5}$$

$$\text{and } u \upharpoonright B + B'' \in id_B \tag{3.6}$$

$$\text{and } u \upharpoonright C + C' \in id_C \tag{3.7}$$

$$\text{then } u \upharpoonright A + B' + C' = \psi^{-1}(t) \tag{3.8}$$

Let (3.4), (3.5), (3.6) and (3.7) be true. Let us denote $\psi^{-1}(t)$ by s. To prove (3.8), we need to show a bijection $\alpha : E_s \to E_{u \upharpoonright A+B'+C'}$ such that $(\forall e_1, e_2 \in E_s)(e_1 \preceq_s e_2 \text{ iff } \alpha(e_1) \preceq_{u \upharpoonright A+B'+C'} \alpha(e_2))$ and $\lambda_s = \lambda_{u \upharpoonright A+B'+C'} \circ \alpha$. We first define a bijection $\phi : E_t \to E_{u \upharpoonright A+B'+C'}$ as follows. For all $e \in E_t$,

- if $\lambda_t(e) \in inl(L_A)$, then $\phi(e) = e$,
- if $\lambda_t(e) \in inr(inl(L_B))$, then there is $e'' \in E_u$ such that $\phi(e) = e_1$ and $e_1 \approx_u e'' \approx_u e$ and $\lambda_u(e) = inl(inr(inl(inl(b))))$ and $\lambda_u(e'') = inl(inr(inr(b)))$ and $\lambda_u(e_1) = inl(inl(inr(b)))$, $b \in L_B$,
- if $\lambda_t(e) \in inr(inr(L_C))$, then $\phi(e) = e_2$ where $e_2 \approx_u e$ and $\lambda_u(e) = inl(inr(inl(inr(c))))$ and $\lambda_u(e_2) = inr(c)$, $c \in L_C$.

We know ϕ is well-defined because the definition of *id* ensures the existence and singularity ensures the uniqueness of the output. In addition, its inverse ϕ^{-1}, defined as follows for all $e \in E_{u \upharpoonright A+B'+C'}$, is a function.

- If $\lambda_{u \upharpoonright A+B'+C'}(e) \in inl(inl(L_A))$, then $\phi^{-1}(e) = e$.
- If $\lambda_{u \upharpoonright A+B'+C'}(e) \in inl(inr(L_B))$, then there is $e'' \in E_u$ such that $\phi^{-1}(e) = e_1$ and $e_1 \approx_u e'' \approx_u e$ and $\lambda_u(e_1) = inl(inr(inl(inl(b))))$ and $\lambda_u(e'') = inl(inr(inr(b)))$ and $\lambda_u(e) = inl(inl(inr(b)))$ and $b \in L_B$.
- If $\lambda_{u \upharpoonright A+B'+C'}(e) \in inr(L_C)$, then $\phi^{-1}(e) = e_2$ where $e_2 \approx_u e$ and $\lambda_u(e_2) = inl(inr(inl(inr(c))))$ and $\lambda_u(e) = inr(c)$, $c \in L_C$.

Therefore, ϕ is a bijection. Since $E_s = E_t$, it follows $\alpha = \phi \circ id$ is a bijection. We can show that $e_1 \preceq_t e_2$ iff $\phi(e_1) \preceq_{u\restriction A+B'+C'} \phi(e_2)$ using a case study like the one in the proof of Lemma 3.24. Since $\preceq_s = \preceq_t$ and $E_s = E_t$ we get $e_1 \preceq_s e_2$ iff $\alpha(e_1) \preceq_{u\restriction A+B'+C'} \alpha(e_2)$.

Next, we show that $\lambda_s = \lambda_{u\restriction A+B'+C'} \circ \alpha$. First, note that domains of the two functions are bijective and the codomains are equal ($L_A + L_B + L_C = L_A + L_{B'} + L_{C'}$). We then prove that for all $e \in E_s$, we have $\lambda_s(e) = \lambda_{u\restriction A+B'+C'} \circ \alpha(e)$.

Let $\lambda_s(e) = inl(inl(a)), a \in L_A$. So, $\lambda_t(e) = inl(a)$. We have $\phi(e) = e$. Since $u \restriction A+B+C = t$, we have $\lambda_u(e) = inl(inl(inl(a)))$. Hence, $\lambda_{u\restriction A+B'+C'}(e) = inl(inl(a))$.

Let $\lambda_s(e) = inl(inr(b)), b \in L_B$. So, $\lambda_t(e) = inr(inl(b))$. We have $\phi(e) = e_1$ such that $e \approx_u e_1$ and $\lambda_u(e) = inl(inr(inl(inl(b))))$ and $\lambda_u(e_1) = inl(inl(inr(b)))$. We then have $\lambda_{u\restriction A+B'+C'}(e_1) = inl(inr(b))$.

Let $\lambda_s(e) = inr(c), c \in L_C$. So, $\lambda_t(e) = inr(inr(c))$. We have $\phi(e) = e_2$ such that $e \approx_u e_2$ and $\lambda_u(e) = inl(inr(inl(inr(c))))$ and $\lambda_u(e_2) = inr(c)$. We then have $\lambda_{u\restriction A+B'+C'}(e_2) = inr(c)$.

Now we prove $(\Lambda_{A,B,C}(\sigma) \otimes id_B); eval_{B,C} \supseteq \sigma$. Let $s \in \sigma$. We will find an interaction trace $u \in int(A \otimes B', (B \Rightarrow C) \otimes B'', C')$ such that $u \restriction A+B+C = \psi^{-1}(s) = t$ and $u \restriction B+B'' \in id_B$ and $u \restriction B'+B'' \in id_B$ and $u \restriction C+C' \in id_C$ and $u \restriction A+B'+C' = s$. Let u be defined as follows.

- $E_u = E_{s\restriction B} + E_{s\restriction A} + E_{s\restriction B} + E_{s\restriction B} + E_{s\restriction C} + E_{s\restriction C}$. Let us refer to these subsets using injections in_1 to in_6.
- $E_u = \lambda_{s\restriction B} + \lambda_{s\restriction A} + \lambda_{s\restriction B} + \lambda_{s\restriction B} + \lambda_{s\restriction C} + \lambda_{s\restriction C}$.
- The preorder $\preceq_u$ is defined as follows. For all $e, e' \in E_s$,

  P1 if $e, e' \in E_{s\restriction A}$, then $e \preceq_s e' \Leftrightarrow in_2(e) \preceq_u in_2(e')$

  P2 if $e, e' \in E_{s\restriction B}$, then $e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_3(e') \Leftrightarrow in_1(e) \preceq_u in_4(e') \Leftrightarrow in_3(e) \preceq_u in_1(e') \Leftrightarrow in_3(e) \preceq_u in_3(e') \Leftrightarrow in_3(e) \preceq_u in_4(e') \Leftrightarrow in_4(e) \preceq_u in_1(e') \Leftrightarrow in_4(e) \preceq_u in_3(e') \Leftrightarrow in_4(e) \preceq_u in_4(e')$,

  P3 if $e, e' \in E_{s\restriction C}$, then $[e \preceq_s e' \Leftrightarrow in_5(e) \preceq_u in_5(e')$ and $in_6(e) \preceq_u in_6(e')$ and $in_5(e) \preceq_u in_6(e')$ and $in_6(e) \preceq_u in_5(e')]$

  P4 if $e \in E_{s\restriction A}$ and $e' \in E_{s\restriction B}$, then $[e \preceq_s e' \Leftrightarrow in_2(e) \preceq_u in_1(e') \Leftrightarrow in_2(e) \preceq_u in_3(e') \Leftrightarrow in_2(e) \preceq_u in_4(e')]$

P5 if $e \in E_{s\restriction A}$ and $e' \in E_{s\restriction C}$, then $[e \preceq_s e' \Leftrightarrow in_2(e) \preceq_u in_5(e') \Leftrightarrow in_2(e) \preceq_u in_6(e')]$

P6 if $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $[e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_3(e) \preceq_u in_2(e') \Leftrightarrow in_4(e) \preceq_u in_2(e')]$

P7 if $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction C}$, then $[e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_5(e') \Leftrightarrow in_1(e) \preceq_u in_6(e') \Leftrightarrow in_3(e) \preceq_u in_5(e') \Leftrightarrow in_3(e) \preceq_u in_6(e') \Leftrightarrow in_4(e) \preceq_u in_5(e') \Leftrightarrow in_4(e) \preceq_u in_6(e')]$

P8 if $e \in E_{s\restriction C}$ and $e' \in E_{s\restriction A}$, then $[e \preceq_s e' \Leftrightarrow in_5(e) \preceq_u in_2(e') \Leftrightarrow in_6(e) \preceq_u in_2(e')]$

P9 if $e \in E_{s\restriction C}$ and $e' \in E_{s\restriction B}$, then $[e \preceq_s e' \Leftrightarrow in_5(e) \preceq_u in_1(e') \Leftrightarrow in_5(e) \preceq_u in_3(e') \Leftrightarrow in_5(e) \preceq_u in_4(e') \Leftrightarrow in_6(e) \preceq_u in_1(e') \Leftrightarrow in_6(e) \preceq_u in_3(e') \Leftrightarrow in_6(e) \preceq_u in_4(e')]$

We use the same strategy as the proof of Lemma 3.24 to show that $\preceq_u$ is a total preorder and u respect singularity. We can also check, following the structure of the proof of Lemma 3.24 that u respects the conditions set out above. ■

CHAPTER 4

COMPOSITIONAL ROUND ABSTRACTION*

Having introduced, in Chapter 3, a model of synchronous behaviours, we return to a question we asked in Chapter 2: how to derive synchronous processes from asynchronous ones?

In [BCG99], Benveniste *et al.* demonstrated that even recovering synchrony, after it is removed, is nontrivial. A more general answer appears in the work of Alur and Henzinger. *Round abstraction* is a technique allowing synchronous systems to be built from asynchronous ones by aggregating serialised events into more abstract macro-steps.

We will build on this notion by reformulating it within our compositional trace model. Our choice is motivated by the generality and simplicity of round abstraction: by aggregating events together, we can form lower-latency processes. In particular, when these processes stem from asynchronous processes, they can also be viewed as locally-synchronous processes.

We begin with an overview of the original formulation of round abstraction in Section 4.1. Then, in Section 4.2, we recast round abstraction within the trace model of Chapter 3. We demonstrate, using examples, that round abstraction is not compositional in general. We then seek sufficient conditions to guarantee compositionality in Section 4.3. We finish with a discussion in Section 4.5.

4.1 Alur-Henzinger Round Abstraction

Reactive Modules (RM) [AH99] is a specification language that supports a round-based form of synchrony: computation proceeds in a sequence of globally-synchronised steps during which variables are updated by both the environment and the system.

*Extended and revised version of [GM10]

The basic computational unit, called a reactive module, consists of a *declaration* and a *body*. The former is a set of typed variables partitioned into,

- a subset of *private variables*, writable to the module but hidden from the environment,
- a subset of *interface variables*, writable to the module and readable to the environment,
- a subset of *external variables*, readable to the module and writable to the environment.

The body is a set of *atoms*, each pairing a set of variables with some guarded behaviour, subject to some consistency and acyclicity conditions. Module execution starts by a single *initialisation round* where all private and interface variables are assigned values. This is followed by *update rounds* during which, the external variables are assigned values and then the atoms of the module are run in order.

As a simple example, consider the toy module in Figure 4.1 specifying a counter, where latched (or updated) values are denoted by primed variable names. The first atom counts the number of rounds (or global clock cycles). The second atom counts the number of rounds when a certain signal (*tick* represented by a boolean variable) is present. The last atom issues a signal (*eq*) whenever the first counter holds a value that is twice the value of the second.

Reactive Modules introduces the notion of abstraction both on the space and the time axes. Spatial abstraction essentially amounts to hiding implementation details; for example, a system may be built from smaller components but appears atomic. Temporal abstraction hides *computational steps*, such that a computation may consist of several steps but appears atomic to an external observer. Moreover, RM provides the means to move along both axes. Consequently, it is possible to convert a synchronous system to an asynchronous one using *variable hiding* to introduce *stuttering*, and conversely, using *round abstraction*. Only the latter will be of interest to us.

Round abstraction is a technique that allows temporal scaling by introducing a variable notion of what constitutes a *computational step*. It has important precursors in the notion of *multiform time* in synchronous languages [BLJ91, Hal93]; and work on *action refinement* [AH89, AH94, GGR94, GR01], *abstract interpretation* [CC77], and *clock variables* [AH97b]. It allows the aggregation of many computational steps into a single macro-step. In Reactive Modules, this

module *Counter*
  external *tick* : $\mathbb{B}$
  interface *eq* : $\mathbb{B}$
  private *roundcount*, *tickcount* : $\mathbb{N}$
  atom *roundcount* **reads** *roundcount*
    init
      $[]\ true \rightarrow roundcount' := 0$
    update
      $[]\ true \rightarrow roundcount' := roundcount + 1$
  atom *tickcount* **reads** *tickcount* **awaits** *tick*
    init
      $[]\ true \rightarrow tickcount' := 0$
    update
      $[]\ tick? \rightarrow tickcount' := tickcount + 1$
  atom *eq* **reads** *roundcount*, *tickcount*
    init update
      $[]\ roundcount = tickcount * 2 \rightarrow eq!$

Figure 4.1: A *Counter* in Reactive Modules

is achieved using the operator **next**: given a module P and a set Y of interface variables, the module **next** Y **for** P is one where all consecutive rounds between two changes in any element of Y are collapsed into a single round. For example, a module behaving like **next** *eq* **for** *Counter* is depicted in Figure 4.2.

One can think of round abstraction in slightly broader terms [AHR98]: if M is a system specification, and ϕ is a condition on the variables of M, then the round abstraction **next** ϕ **for** M gathers as many transitions of M as necessary to satisfy ϕ, and *hides the intermediate steps*.

In [AW99], Alur and Wang study round abstraction as a heuristic for model checking. In the same vein as [AHR98], the paper exploits the fact that round abstraction typically results in fewer states. Subsequently, it seeks to establish a congruence result between a process and its round abstraction. This turns out to hold when round abstraction does not aggregate transitions that access shared variables. We will encounter a similar problem in Section 4.5.

module *RACounter*
  external *tick* : $\mathbb{B}$
  interface *eq* : $\mathbb{B}$
  private *roundcount*, *tickcount* : $\mathbb{N}$
  atom *eq* **reads** *roundcount*, *tickcount*
    init update
      $[]\ roundcount = tickcount * 2 \rightarrow eq!$

Figure 4.2: A round abstracted *Counter*

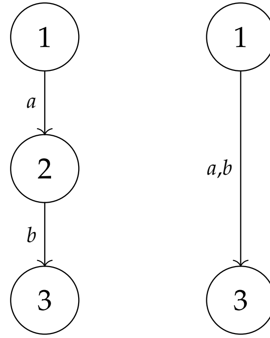


Figure 4.3: Round abstraction on automata

Round abstraction is interesting because it defines a simple technique for constructing synchronous systems from asynchronous ones. Indeed it introduces a clock, such that the notion of a round is solely based on the input/output behaviour of systems. However, the original formulation of round abstraction is monolithic and applies to whole systems, not addressing the question of whether round-abstracted systems still interact correctly with each other.

In the next section, we will proceed by defining round abstraction within our trace model of processes. We will avoid the choice of a clock, which is arbitrary, and thus provide a more general notion of round abstraction. In particular, we will discuss a round abstraction where the intermediate events are not hidden. We therefore consider round abstraction as an approximation technique which removes some of the timing information between events in a process. In the lack of a global clock, our definition of round abstraction resembles so-called polychronous operators in SIGNAL [BLJ91, LTL03].

4.2 Round Abstraction on Processes

Our goal is to develop a notion of round abstraction that can be applied to asynchronous processes compositionally. In other words, we would like to establish, for processes $\sigma : A \to B$ and $\tau : B \to C$, that if σ' is a round abstraction of σ and τ' is a round abstraction of τ, then $\sigma';\tau'$ is a round abstraction of $\sigma;\tau$.

We start with the simple notion of round abstraction of traces, which we obtain by gathering serialised events in coarser-grained *rounds* that we interpret as consisting of simultaneous events.

At this juncture, it may be observed that locally-synchronous traces have an inherent *order of synchronicity*. For example, for each possible succession of unique events, the least synchronous trace is the one where they occur one after the other, while the most synchronous trace is the one where they all occur simultaneously.

Definition 4.1 (Round abstraction on traces). *Let $s = \langle E_s, \preceq_s, \lambda_s \rangle$, $t = \langle E_t, \preceq_t, \lambda_t \rangle$ be traces on A. We say that t* is a round abstraction of *s, written $s \sqsubseteq t$, if there exists a bijection $\phi : E_s \to E_t$ such that $\langle E_s, \lambda_s \rangle$ and $\langle E_t, \lambda_t \rangle$ are ϕ-isomorphic, i.e. $\lambda_s = \lambda_t \circ \phi$, and ϕ is monotonic relative to temporal ordering, i.e. for any $e, e' \in E_s$, if $e \preceq_s e'$, then $\phi(e) \preceq_t \phi(e')$.*

It follows from the definition that simultaneity is preserved; that is, if $e \approx_s e'$, then $\phi(e) \approx_t \phi(e')$. The converse is obviously false, since round abstraction can make non-simultaneous events in s simultaneous in t.

The next step is to lift the definition of round abstraction to processes. In order to appreciate the challenges we need to address, let us first informally introduce two definitions of round abstraction on processes, which we shall call *partial* and *total*. Partial round abstraction, $\sigma \sqsubseteq \tau$, requires that τ, the 'abstracted' process, has no 'junk' traces which do not stem from σ. A stronger property, total round abstraction, written $\sigma \precapprox \tau$, additionally requires that all the behaviour of σ can be found, in an abstracted form, in τ. In general, it is not the case that $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ implies $\sigma;\tau \sqsubseteq \sigma';\tau'$, or $\sigma \precapprox \sigma'$ and $\tau \precapprox \tau'$ implies $\sigma;\tau \precapprox \sigma';\tau'$. This situation is akin to the non-compositionality of *abstract interpretation* [Abr90]. As immediate counter-examples, consider the following.

Example 4.2. Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$ be processes, with $L_A = \{a\}$, $L_B = \{b_1, b_2\}$, $L_C = \{c\}$, defined as follows.

$$\sigma = pc(\{b_2.b_1.a\}) \qquad \sqsubseteq \qquad \sigma' = pc(\{\langle b_2, b_1 \rangle.a\})$$
$$\tau = pc(\{c.b_1.b_2\}) \qquad \sqsubseteq \qquad \tau' = pc(\{c.\langle b_1, b_2 \rangle\})$$

We then have $\sigma;\tau = pc(\{c\})$ but $\sigma';\tau' = pc(\{c.a\}) \not\sqsupseteq \sigma;\tau$.

Example 4.3. Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$ be processes, with $L_A = \{a\}$, $L_B = \{b_1, b_2\}$,

$L_C = \{c\}$, defined as follows.

$$\sigma = pc(\{b_1.b_2.a\}) \quad \sqsubseteq\!\!\!\approx \quad \sigma' = pc(\{\langle b_1, b_2\rangle.a\})$$

$$\tau = pc(\{c.b_1.b_2\}) \quad \sqsubseteq\!\!\!\approx \quad \tau' = pc(\{c.b_1.b_2\})$$

Then, we have $\sigma;\tau = pc(\{c.a\})$ but $\sigma';\tau' = \{c\} \not\sqsupseteq\!\!\!\approx \sigma;\tau$.

In these examples, and typically, the way *deadlock* is handled will play the key role, because round abstraction can both resolve and introduce deadlocks. In Example 4.2, the two processes do not compose because the order in which b_1, b_1 are issued by σ does not coincide with the order in which they can be received by τ; round abstraction makes the two events simultaneous and thus solves the deadlock. In Example 4.3, round abstraction requires the two B-events to be simultaneous in σ' and consecutive in τ' thereby introducing deadlock.

In the next section, we formalise partial round abstraction, then seek sufficient conditions to guarantee compositionality.

4.3 Compositionality of Partial Round Abstraction

We define partial round abstraction on processes as follows.

Definition 4.4 (Partial round abstraction). *For processes σ and τ over A, we say that τ is a* partial round abstraction *of σ, written $\sigma \sqsubseteq\!\!\!\approx \tau$, if for any $t \in \tau$ there is $s \in \sigma$ such that $s \sqsubseteq\!\!\!\approx t$.*

In a partial round abstraction, the abstracted process does not contain any 'junk' traces which do not correspond to traces in the original process. However, it is possible for some traces in the original process to have no corresponding trace in the abstraction.

The following technical lemma and its corollary will be useful in later proofs.

Lemma 4.5. *For traces s, s' over signature $A + B$, if $s \sqsubseteq\!\!\!\approx s'$, then $s \upharpoonright A \sqsubseteq\!\!\!\approx s' \upharpoonright A$.*

Proof. Projection does not affect the temporal order of those events that remain after projection. By definition, $s \sqsubseteq\!\!\!\approx s'$ implies that $E_s = E_{s'}$ and $\lambda_s = \lambda_{s'}$. So, $\{e \in E_s \mid \lambda_s(e) \in lin(L_A)\} = \{e' \in E_{s'} \mid \lambda_{s'}(e') \in lin(L_A)\}$; that is, $E_{s\upharpoonright A} = E_{s'\upharpoonright A}$. Since $\preceq_s \subseteq \preceq_{s'}$ and using Lemma 3.19, it follows that $\preceq_{s\upharpoonright A} \subseteq \preceq_{s'\upharpoonright A}$. ∎

Corollary 4.6. *For processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \lightning \tau \sqsubseteq \sigma' \lightning \tau'$, then $\sigma ; \tau \sqsubseteq \sigma' ; \tau'$.*

A trace is a permutation of another if it has the 'same events' irrespective of order.

Definition 4.7 (Permutation)**.** *Traces s and t over A are* permutations *of each other, written $s \sim t$, if there exists a bijection $\phi : E_s \to E_t$ satisfying $\lambda_s = \lambda_t \circ \phi$.*

Given a trace v over A, let $\Pi(v)$ be the set of its permutations.

In order to prevent round abstraction from resolving deadlocks, as in Example 4.2, we introduce the following condition.

Definition 4.8 (Compatibility)**.** *Processes $\sigma : A \to B$ and $\tau : B \to C$ are* compatible, *written $\sigma \asymp \tau$, when the following conditions hold.*

1. *For any $s \in \sigma$, and any $u \in int(A, B, C)$ such that $u \upharpoonright B + C \in \tau$, if*
   - *$u \upharpoonright B \in \Pi(s \upharpoonright B)$*
   - *and $u \upharpoonright A = s \upharpoonright A$*

   then $u \upharpoonright A + B \in \sigma$.

2. *For any $t \in \tau$, and any $u \in int(A, B, C)$ such that $u \upharpoonright A + B \in \sigma$, if*
   - *$u \upharpoonright B \in \Pi(t \upharpoonright B)$*
   - *and $u \upharpoonright C = t \upharpoonright C$*

   then $u \upharpoonright B + C \in \tau$.

Compatibility ends up ensuring compositionality for partial round abstraction almost by definition. Its merit is rather as a characterisation of the main cause of failure of composition for partial round abstraction. Going back to Example 4.2, the composition of $b_2.b_1.a \in \sigma$ and $c.b_1.b_2 \in \tau$ fails because they produce the same B-events in different orders.

Nevertheless, in some contexts, compatibility may be too strong a requirement. For example, the game model of [GM08] does not guarantee that the strategies in Figure 4.4 are compatible.

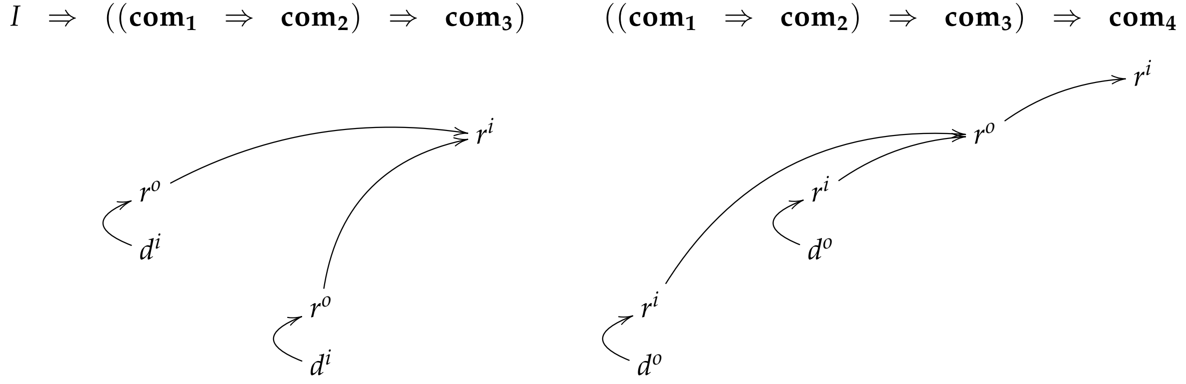


Figure 4.4: When compatibility is too strong

We therefore introduce a weaker version of compatibility which takes round abstraction into account. Instead of requiring processes not to deadlock in composition, we only require traces whose round abstractions compose not to cause deadlock.

Definition 4.9 (Post-compatibility). *Processes $\sigma' : A \to B$ and $\tau' : B \to C$ are* post-compatible *with respect to processes $\sigma : A \to B$ and $\tau : B \to C$, respectively, written $\sigma' \bigcirc \tau'$, when the following conditions hold.*

1. *For any $s \in \sigma$ and any $u \in int(A, B, C)$ such that $u \upharpoonright B + C \in \tau$, if*
   - *s has a round abstraction $s' \in \sigma'$ and $u \upharpoonright B + C$ has a round abstraction $t' \in \tau'$, such that $s' \upharpoonright B = t' \upharpoonright B$*
   - *and $u \upharpoonright A = s \upharpoonright A$*

   then $u \upharpoonright A + B \in \sigma$.

2. *For any $t \in \tau$ and any $u \in int(A, B, C)$ such that $u \upharpoonright A + B \in \sigma$, if*
   - *$u \upharpoonright A + B$ has a round abstraction $s' \in \sigma'$ and t has a round abstraction $t' \in \tau'$, such that $s' \upharpoonright B = t' \upharpoonright B$*
   - *and $u \upharpoonright C = t \upharpoonright C$*

   then $u \upharpoonright B + C \in \tau$.

One of our main results is the soundness of composition relative to partial round abstraction. It is guaranteed by either compatibility or post-compatibility.

Lemma 4.10. *Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$ be processes such that $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$. For all $u' \in \sigma' \lightning \tau'$, there are interaction traces $u, v \in int(A, B, C)$ and traces $s \in \sigma$ and $t \in \tau$ satisfying,*

1. *$u \sqsubseteq u'$ and $u \upharpoonright A + B = s$ and $u \upharpoonright B \in \Pi(t \upharpoonright B)$ and $u \upharpoonright C = t \upharpoonright C$*
2. *$v \sqsubseteq u'$ and $v \upharpoonright B + C = t$ and $v \upharpoonright B \in \Pi(s \upharpoonright B)$ and $v \upharpoonright A = s \upharpoonright A$.*

Proof. Let $u' \in \sigma' \lightning \tau'$; that is, $u' \upharpoonright A + B \in \sigma'$ and $u' \upharpoonright B + C \in \tau'$. Since $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$, it follows there are $s \in \sigma$ and $t \in \tau$ such that $s \sqsubseteq u' \upharpoonright A + B$ and $t \sqsubseteq u' \upharpoonright B + C$. So, let $\phi : E_s \to E_{u' \upharpoonright A+B}$ and $\psi : E_t \to E_{u' \upharpoonright B+C}$ be bijections such that $\lambda_s = \lambda_{u' \upharpoonright A+B} \circ \phi$ and $\lambda_t = \lambda_{u' \upharpoonright B+C} \circ \psi$. Without loss of generality, we will assume that ϕ and ψ are identities, implying that $E_s = E_{u' \upharpoonright A+B}$ and $E_t = E_{u' \upharpoonright B+C}$. It follows that $E_{s \upharpoonright B} = E_{t \upharpoonright B}$ and so $s \upharpoonright B$ is a permutation of $t \upharpoonright B$.

The interaction trace u is defined as follows.

- $E_u = E_{u'}$
- $\lambda_u = \lambda_{u'}$
- We define $\preceq_u$ as follows. First, note that for any preorder $\preceq$, there is an associated strict partial order $\prec$, defined as $a \prec b$ if and only if $a \preceq b$ and $b \not\preceq a$. Let the relations $\preceq_1 \subseteq \preceq_{u'}$ and $\preceq_2 \subseteq \preceq_{u'}$ be defined as follows, for all $e, e' \in E_u$.

  P1 $e \prec_{u'} e' \Leftrightarrow e \preceq_1 e'$.

  P2 If $e, e' \in E_s$, then $e \preceq_s e' \Leftrightarrow e \preceq_2 e'$.

  P3 If $e, e' \in E_{t \upharpoonright C}$, then $e \preceq_t e' \Leftrightarrow e \preceq_2 e'$.

  P4 If $e \in E_{t \upharpoonright C}$ and $e' \in E_{s \upharpoonright B}$, then $e \approx_{u'} e' \Leftrightarrow e \preceq_2 e'$.

  P5 If $e \in E_{t \upharpoonright C}$ and $e' \in E_{s \upharpoonright A}$, then $e \approx_{u'} e' \Leftrightarrow e \preceq_2 e'$.

  P6 If $e \in E_s$ and $e' \in E_{t \upharpoonright C}$, then $e \not\preceq_2 e'$.

  We then set $\preceq_u = \preceq_1 \cup \preceq_2$.

We first show that $\preceq_u$ is a total preorder. Since $\preceq_s$ and $\preceq_t$ are reflexive, (P2) and (P3) ensure that $\preceq_u$ is reflexive. Next, we prove that $\preceq_u$ is total, i.e. $(\forall e, e' \in E_u)(e \preceq_u e'$ or $e' \preceq_u e)$

using a case analysis. Take $e, e' \in E_u$. There are nine cases corresponding to the possible label assignments of e and e'. We abuse notation by omitting label injections.

1. If $[\lambda_u(e) \in L_A$ or $\lambda_u(e) \in L_B]$ and $[\lambda_u(e') \in L_A$ or $\lambda_u(e') \in L_B]$, then $e \preceq_s e'$ or $e' \preceq_s e$ since $\preceq_s$ is total. Using (P2), we get $e \preceq_u e'$ or $e' \preceq_u e$.

2. If $\lambda_u(e) \in L_C$ and $\lambda_u(e') \in L_C$, we use the same strategy as the first case with (P3) and the fact that $\preceq_t$ is total.

3. If $[\lambda_u(e) \in L_C$ and $\lambda_u(e') \in L_B]$ or $[\lambda_u(e) \in L_B$ and $\lambda_u(e') \in L_C]$, then given that $\preceq_{u'}$ is total, we have $e \preceq_{u'} e'$ or $e' \preceq_{u'} e$. We can rewite this as $e \prec_{u'} e'$ or $e' \prec_{u'} e$ or $e \approx_{u'} e'$. In the first two cases, we use (P1). In the third case, we (P4).

4. If $[\lambda_u(e) \in L_C$ and $\lambda_u(e') \in L_A]$ or $[\lambda_u(e) \in L_A$ and $\lambda_u(e') \in L_C]$, then given that $\preceq_{u'}$ is total, we have $e \preceq_{u'} e'$ or $e' \preceq_{u'} e$. We can rewite this as $e \prec_{u'} e'$ or $e' \prec_{u'} e$ or $e \approx_{u'} e'$. In the first two cases, we use (P1). In the third case, we (P5).

We now show that $\preceq_u$ is transitive, i.e. $(\forall e, e', e'' \in E_u)(\text{if } e \preceq_u e' \text{ and } e' \preceq_u e'', \text{then } e \preceq_u e'')$. Let $e \preceq_u e'$ and $e' \preceq_u e''$. We have the following four cases.

1. If $e \preceq_1 e'$ and $e' \preceq_1 e''$, then $e \prec_{u'} e'$ and $e' \prec_{u'} e''$. Since $\prec_{u'}$ is a strict partial order, we get $e \prec_{u'} e''$. Using (P1), we get $e \preceq_u e''$.

2. If $e \preceq_1 e'$ and $e' \preceq_2 e''$, then $e \prec_{u'} e'$ and $e' \preceq_{u'} e''$. Expanding the definition of $\prec_{u'}$, we get $e \preceq_{u'} e'$ and $e' \not\preceq_{u'} e$ and $e' \preceq_{u'} e''$. Since $\preceq_{u'}$ is transitive, we get $e \preceq_{u'} e''$. Suppose to the contrary that $e'' \preceq_{u'} e$. Then, as $\preceq_{u'}$ is transitive, we get $e' \preceq_{u'} e$ which contradicts $e' \not\preceq_{u'} e$. So $e'' \not\preceq_{u'} e$. We conclude that $e \prec_{u'} e''$ then use (P1) to get $e \preceq_u e''$.

3. If $e \preceq_2 e'$ and $e' \preceq_1 e''$, then we use the same strategy as the second case.

4. If $e \preceq_2 e'$ and $e' \preceq_2 e''$, then we consider the 27 cases corresponding to all possible label assignments to the events e, e', e''. For the sake of succinctness, we will denote label assignments by a triple. So we write (X, Y, Z) to mean that e is an X-event, e' is a Y-event and e'' is a Z-event.

   - Cases $(A, A, A), (A, A, B), (A, B, A), (A, B, B), (B, A, A), (B, A, B), (B, B, A), (B, B, B)$. We use (P2) and the fact that $\preceq_s$ is transitive.

- Case (C,C,C). We use (P3) and the fact that $\preceq_t$ is transitive.
- Cases $(A,A,C),(A,B,C),(A,C,A),(A,C,B),(A,C,C),(B,A,C),(B,B,C),(B,C,B),(B,C,A),(B,C,C),(C,A,C),(C,B,C)$ are not possible by definition.
- Cases $(C,A,A),(C,A,B),(C,B,A),(C,B,B)$. From $e \preceq_2 e'$ and $e' \preceq_2 e''$, we get $e \approx_{u'} e'$ and $e' \preceq_s e'' \therefore e' \preceq_{u'} e''$. Since $\preceq_{u'}$ is total, we have either
  - $e'' \preceq_{u'} e$, which yields $e \approx_{u'} e''$ since $\preceq_{u'}$ is transitive. Then, depending on the specific case, we use (P4) or (P5) to get $e \preceq_u e''$.
  - $e'' \not\preceq_{u'} e$, so $e \prec_{u'} e''$. By (P1), $e \preceq_u e''$.
- Cases $(C,C,A),(C,C,B)$. From $e \preceq_2 e'$ and $e' \preceq_2 e''$, we get $e \preceq_t e' \therefore e \preceq_{u'} e'$ and $e' \approx_{u'} e''$. Since $\preceq_{u'}$ is total, we have either
  - $e'' \preceq_{u'} e$, which yields $e \approx_{u'} e''$ since $\preceq_{u'}$ is transitive. Then, depending on the specific case, we use (P4) or (P5) to get $e \preceq_u e''$.
  - $e'' \not\preceq_{u'} e$, so $e \prec_{u'} e''$. By (P1), $e \preceq_u e''$.

The trace u has singular events because u' respects singularity and $\preceq_u \subseteq \preceq_{u'}$.

Next, we show that u satisfies the conditions outlined in the lemma.

- First, we show that $u \sqsubseteq u'$. This follows from the following facts: $E_u = E_{u'}$, $\lambda_u = \lambda_{u'}$ and $\preceq_u \subseteq \preceq_{u'}$.
- Then, we prove that $u \upharpoonright B \in \Pi(t \upharpoonright B)$. This follows from $E_{u\upharpoonright B} = E_{u'\upharpoonright B} = E_{t\upharpoonright B}$ and $\lambda_{u\upharpoonright B} = \lambda_{u'\upharpoonright B} = \lambda_{t\upharpoonright B}$.
- Next, we need to show that $u \upharpoonright A+B = s$. We have $E_{u\upharpoonright A+B} = E_{u'\upharpoonright A+B} = E_s$ and $\lambda_{u\upharpoonright A+B} = \lambda_{u'\upharpoonright A+B} = \lambda_s$. We need to show that $\preceq_s = \preceq_{u\upharpoonright A+B}$. The left to right inclusion follows from (P2). For the right to left inclusion, take $e, e' \in E_{u\upharpoonright A+B}$ such that $e \preceq_u e'$. Using the definition of $\preceq_u$, we get either $e \preceq_s e'$ or $e \prec_{u'} e'$. In the first case, we are done. Expanding the definition of the second case, we get $e \preceq_{u'} e'$ and $e' \not\preceq_{u'} e$. Since $\preceq_s$ is total, we have $e \preceq_s e'$ or $e' \preceq_s e$. We also have that $\preceq_s \subseteq \preceq_{u'}$. However, since $e' \not\preceq_{u'} e$ we get $e' \not\preceq_s e$. So $e \preceq_s e'$.
- Finally, we can demonstrate that $u \upharpoonright C = t \upharpoonright C$ using the same strategy used to show that $u \upharpoonright A+B = s$.

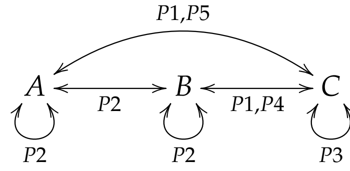


Figure 4.5: Totality of $\preceq_u$ in Lemma 4.10

The second part of the proof is symmetric. So, we only give the definition of the interaction trace v.

- $E_v = E_{u'}$
- $\lambda_v = \lambda_{u'}$
- We define $\preceq_v$ as follows. Let the relations $\preceq_3 \subseteq \preceq_{u'}$ and $\preceq_4 \subseteq \preceq_{u'}$ be defined as follows, for all $e, e' \in E_u$.
    1. $e \prec_{u'} e' \Leftrightarrow e \preceq_3 e'$.
    2. If $e \in E_t$ and $e' \in E_t$, then $e \preceq_t e' \Leftrightarrow e \preceq_4 e'$.
    3. If $e, e' \in E_{s\restriction A}$, then $e \preceq_s e' \Leftrightarrow e \preceq_4 e'$.
    4. If $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $e \approx_{u'} e' \Leftrightarrow e \preceq_2 e'$.
    5. If $e \in E_{t\restriction C}$ and $e' \in E_{s\restriction A}$, then $e \approx_{u'} e' \Leftrightarrow e \preceq_2 e'$.

  We then set $\preceq_v = \preceq_3 \cup \preceq_4$. ∎

Theorem 4.11 (Soundness I). *For any processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\sigma \asymp \tau$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

Proof. By Corollary 4.6, it is sufficient to show that $\sigma \between \tau \sqsubseteq \sigma' \between \tau'$. Let $u' \in \sigma' \between \tau'$. By Lemma 4.10, there is a trace $t \in \tau$ and an interaction trace $u_1 \in int(A, B, C)$ satisfying $u_1 \sqsubseteq u'$ and $u_1 \restriction A + B \in \sigma$ and $u_1 \restriction B \in \Pi(t \restriction B)$ and $u_1 \restriction C = t \restriction C$. Using compatibility, we deduce that $u_1 \restriction B + C$ and hence $u \in \sigma \between \tau$. Since $u_1 \sqsubseteq u'$ we are done. Note that the proof can also be done symmetrically using σ instead of τ. ∎

Theorem 4.12 (Soundness II). *For any processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\sigma' \bigcirc \tau'$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

Proof. By Corollary 4.6, it is sufficient to show that $\sigma \between \tau \sqsubseteq \sigma' \between \tau'$. Let $u' \in \sigma' \between \tau'$. By Lemma 4.10, there are traces $s \in \sigma$, $t \in \tau$ and interaction traces $u_1, u_2 \in int(A, B, C)$ satisfying

- $u_1 \sqsubseteq u'$ and $u_1 \upharpoonright A + B = s$ and $u_1 \upharpoonright B \in \Pi(t \upharpoonright B)$ and $u_1 \upharpoonright C = t \upharpoonright C$
- $u_2 \sqsubseteq u'$ and $u_2 \upharpoonright B + C = t$ and $u_2 \upharpoonright B \in \Pi(s \upharpoonright B)$ and $u_2 \upharpoonright A = s \upharpoonright A$

Since $u_1 \sqsubseteq u'$ and $u_1 \upharpoonright A + B = s$, it follows, by Lemma 4.5, that $s \sqsubseteq u' \upharpoonright A + B$. Since $u_2 \sqsubseteq u'$ and $u_2 \upharpoonright B + C = t$, it follows, by Lemma 4.5, that $t \sqsubseteq u' \upharpoonright B + C$. Using post-compatibility, we get $u_1 \upharpoonright B + C \in \tau$ and $u_2 \upharpoonright A + B \in \sigma$. So, $u_1, u_2 \in \sigma \between \tau$. Since $u_1 \sqsubseteq u'$ and $u_2 \sqsubseteq u'$, we are done. ■

Since tensoring processes interleaves their traces, round abstraction is straightforwardly compositional with respect to the tensor product.

Theorem 4.13 (Soundness III). *For any processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$, then $\sigma \otimes \tau \sqsubseteq \sigma' \otimes \tau'$.*

Proof. Let $v' \in \sigma' \otimes \tau'$; that is, $v' \upharpoonright A + B \in \sigma$ and $v' \upharpoonright C + D \in \tau$. Since $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$, it follows there are traces $s \in \sigma$ and $t \in \tau$ such that $s \sqsubseteq v' \upharpoonright A + B$ and $t \sqsubseteq v' \upharpoonright C + D$.

So, let $\phi : E_s \to E_{v' \upharpoonright A+B}$ and $\psi : E_t \to E_{v' \upharpoonright C+D}$ be bijections such that $\lambda_s = \lambda_{v' \upharpoonright A+B} \circ \phi$ and $\lambda_t = \lambda_{v' \upharpoonright C+D} \circ \psi$ and $(\forall e, e' \in E_s)(\text{if } e \preceq_s e', \text{then } \phi(e) \preceq_{v' \upharpoonright A+B} \phi(e')$ and $(\forall e, e' \in E_t)(\text{if } e \preceq_t e', \text{then } \psi(e) \preceq_{v' \upharpoonright B+C} \psi(e')$.

We show that there is a trace $v \in \sigma \otimes \tau$ such that $v \sqsubseteq v'$. The trace v is constructed as follows.

- $E_v = E_s + E_t$
- $\lambda_v = \lambda_s + \lambda_t$
- Let $\alpha : E_v \to E_{v'} = \phi + \psi$. Let $\preceq_v$ be the relation satisfying
  1. for all $e, e' \in E_v$, if $\alpha(e) \preceq_{v'} \alpha(e')$ and $\alpha(e') \not\preceq_{v'} \alpha(e)$, then $e \preceq_v e'$
  2. for all $e \in E_v$ and for all $e' \in E_v$ such that $\alpha(e) \approx_{v'} \alpha(e')$
     (a) if $e, e' \in E_s$, then $e \preceq_s e' \Leftrightarrow e \preceq_v e'$

(b) if $e, e' \in E_t$, then $e \preceq_t e' \Leftrightarrow e \preceq_v e'$

(c) if $e \in E_s$ and $e' \in E_t$, then $e \preceq_v e'$

Note that in (2c), the choice of ordering is irrelevant. So, we choose to make the events of s before those of t by default. We know that $\preceq_v$ is reflexive because of (1), (2a) and (2b); and transitive and total by construction. Additionally, v has singular events because v' respects singularity and $\preceq_v \subseteq \preceq_{v'}$.

We briefly argue that $v \in \sigma \otimes \tau$. We prove that $v \restriction A + B = s$ and $v \restriction B + C = t$. Let $e, e' \in E_{v \restriction A+B} = E_s$. We show that $e \preceq_{v \restriction A+B} e'$ iff $e \preceq_s e'$. The right to left implication is straightforward. For the left to right implication, let $e \preceq_{v \restriction A+B} e'$. So, $e \preceq_v e'$. Since $v \sqsubseteq v'$, it follows $\alpha(e) \preceq_{v'} \alpha(e')$. This means either

- $\alpha(e) \preceq_{v'} \alpha(e')$ and $\alpha(e') \not\preceq_{v'} \alpha(e) \therefore \phi(e) \preceq_{v' \restriction A+B} \phi(e')$ and $\phi(e') \not\preceq_{v' \restriction A+B} \phi(e) \therefore e \preceq_s e$.
- $\alpha(e) \approx_{v'} \alpha(e')$. Since $e \preceq_v e'$ and $e, e' \in E_s$, it follows that $e \preceq_s e'$.

The proof that $v \restriction B + C = t$ is symmetric. ∎

4.4 Total Round Abstraction

Total round abstraction is a stronger notion of round abstraction: one that not only requires the abstracted process be junk-free, but also that no behaviour be lost. In general, ensuring the compositionality of total round abstraction is more difficult. In this section, we will informally discuss how this may be achieved. Note that we do not introduce any formal results.

Total round abstraction may be defined as follows.

Definition 4.14 (Total round abstraction)**.** *For asynchronous process $\sigma : A \to B$ and process $\sigma' : A \to B$, we say that σ' is a* total round abstraction *of σ, written $\sigma \sqsubseteq\!\!\!\sim \sigma'$, if $\sigma \sqsubseteq \sigma'$ and for any $s \in \sigma$ there exist $w \in \Delta(A \Rightarrow B)$ and $s' \in \sigma'$ such that $s \cdot w \in \sigma$ and $s \cdot w \sqsubseteq s'$.*

Total round abstraction has a more complicated technical definition because prefix-closure is defined at the level of rounds rather than at the level of events. It says that any trace in

the original process can be 'padded' with some events so that it matches a trace in the abstracted process. The reason is that prefix-closure will generate more prefixes for an asynchronous trace than for its synchronous, round abstraction; however, we want round abstraction to automatically extend to prefixes. For example, at the level of traces, $a.b.c \sqsubseteq \langle a,b,c \rangle$ but $pc(a.b.c) = \{\epsilon, a, a.b, a.b.c\}$ whereas $pc(\langle a,b,c \rangle) = \{\epsilon, \langle a,b,c \rangle\}$; using Definition 4.14, we have $pc(a.b.c) \precapprox pc(\langle a,b,c \rangle)$. Note that, unlike partial round abstraction, the above definition is not transitive: from $\sigma \precapprox \sigma'$ and $\sigma' \precapprox \sigma''$, we cannot conclude that $\sigma \precapprox \sigma''$. This fact does not hinder our subsequent discussion, but it presents a challenge that must be addressed in any future solution.

Immediately, there are two problems that seem to hinder compositionality. The first is illustrated by the following example.

Example 4.15. Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, with $L_A = \{a\}$, $L_B = \{b_1, b_2, b_3\}$, $L_C = \{c\}$, be the following processes.

$$\sigma = pc(\{b_1.b_2.a\}) \quad \precapprox \quad \sigma' = pc(\{\langle b_1, b_2, a \rangle\})$$

$$\tau = pc(\{c.b_1.b_3\}) \quad \precapprox \quad \tau' = pc(\{\langle c, b_1, b_3 \rangle\})$$

Then, we have $\sigma;\tau = pc(\{c\})$ but $\sigma';\tau' = \{\epsilon\} \not\succapprox \sigma;\tau$.

In this example, the original processes σ and τ compose well up to b_1 then deadlock as they attempt to synchronise on mismatched events. Because σ' and τ' are single-round processes, the failure of composition prevents the creation of any complete rounds; therefore, it produces only the empty-trace process as a result. A possible solution is to require processes to interact *safely*: if, at any point in the interaction of two processes, one is able to produce an output, then the other must be able to receive it as input.

The second problem was demonstrated in Example 4.3: the possibility that round abstraction may introduce deadlock. This suggests that round abstraction should be subject to restrictions. The following conditions stem immediately from assuming a locally synchronous setting.

Input receptivity: successive inputs can be received in succession as well as simultaneously,

Instant feedback receptivity: an input following an output may also be received simultaneously.

In essence, these rules stipulate that the environment can produce input either instantly or later, and the system must handle both situations. Note that safety and the two receptivity conditions model the requirement that processes must handle all legal inputs from their environment. This may be viewed as an example of Nain and Vardi's Principle of Comprehensive Modelling [NV07]. In fact, Nain and Vardi used the term *receptiveness* to refer to processes that possess this very property [NV07, NV09]. Receptiveness, however, has a long pedigree in computer science. The term was originally coined by Dill in his Ph.D. dissertation [Dil88]. In [AL93], Abadi and Lamport study it as a correctness property of compositional specifications. Similar notions have also appeared in [LT89, AL91, GSSAL94, AH95, AH97a].

In general, further restrictions may be required. For instance, the notion of *interference* [Rey78]—where computations affect each other's outcome—seems of interest to round abstraction. For example, in BSCI, functions do not share identifiers, and therefore do not interfere, with their arguments. By contrast, pairs of arguments may interfere with each other.

Intuitively, events from noninterfering computations may be allowed to occur simultaneously in the round abstraction since their order is arbitrary in any original process [McC02]. However, making interfering events simultaneous is dangerous as it may result in either the round abstraction resolving or introducing deadlocks. We discuss an example of this subtle problem in the next section.

4.5 Discussion

Failure of process compatibility (or post-compatibility) can give rise to subtle problems in synchronous implementations. Let us consider an example from Geometry of Synthesis [Ghi07]. Suppose that an implementer wants to use round abstraction to reduce the latency of (binary) memory locations, as much as possible. Following the game semantic model, these are driven using the ports r (read), t (produce 1), f (produce 0), wt (write 1), wf (write 0), ok (acknowledge write). Singularity prevents multiple reads and multiple writes per round, but one read and one write per round could be implemented. A proper (asynchronous) memory cell trace such

as *wf.ok.r.f.wt.ok.r.t* could be presumably abstracted as $\langle wf, ok, r, f\rangle \cdot \langle wt, ok, r, t\rangle$. This is reasonable, and in fact, assignable variables in certain synchronous languages (e.g. Esterel) and registers in hardware are implemented in this way. However, such an abstraction is incompatible with round abstraction because it breaks process compatibility and therefore partial round abstraction.

The reason is that Basic Syntactic Control of Interference, the language used in GoS, allows asynchronous programs to generate local variable traces that are not consistent with stateful behaviour, but are permutations of well-formed traces. These bad traces are then eliminated via composition with a local variable binder. However, round abstraction may erroneously identify a good trace and a bad trace. For example, an ill-formed trace such as *r.f.wf.ok.r.t.wt.ok* can also be abstracted to $\langle wf, ok, r, f\rangle \cdot \langle wt, ok, r, t\rangle$, which is the same as the abstraction of the well-formed trace above. At the level of the programming language, it means that programs $\mathsf{x} := 0;\ \mathsf{x} := 1;\ \mathsf{if}\ \mathsf{x} = 1\ \mathsf{diverge}$ and $\mathsf{x} := 0;\ \mathsf{if}\ \mathsf{x} = 1\ \mathsf{diverge};\ \mathsf{x} := 1$ could end up with the same implementation, which is obviously erroneous!

Another problem that may occur in total round abstractions is *instant feedback*. Suppose we disallow simultaneous access to variables to overcome the problem above. It follows that a program such as $\mathsf{x} :=!\mathsf{x}$ takes time to run. Further, let us take the round abstraction of the no-op command skip as $\langle r, d\rangle$. This is acceptable, since it agrees with the conditions we set out in this chapter. In fact, Esterel has an equivalent implementation, the instantly terminating command `nothing`. However, in the program $\mathsf{x} :=!\mathsf{x};\ \mathsf{skip}$, the circuit corresponding to sequential composition requires the ability to deal with both commands that take time and those that instantly terminate.

CHAPTER 5

CAUSAL PROCESSES AND ASYNCHRONOUS PROCESSES

Causality is central to many theoretical frameworks including concurrency theory [Win87, Maz95], synchronous languages [Ber99] and component-based systems [GMR10].

In interleaved models of concurrency, the ability to assign an actual cause to each event in a trace is a prerequisite to describing asynchronous behaviour. This is because the order of the occurrence of events must be closed under certain permutations that are not allowed to swap an event and its cause. In higher-level systems, such as games or data flow [GKW85], causality can be encoded directly, as justification pointers or token tags, respectively. Alternatively, in a lower-level system, such as hardware, it is necessary to be able to recover this information implicitly from the structure of the trace [GS10]. In certain game models, justification pointers can also be recovered from the structure of the play [GM03]. We will see in Chapter 6 that causality is also required to describe deterministic synchronous processes.

In the following section, we refine the trace model of Chapter 3 to enable causality to be recorded at the level of the trace. We then describe a category of asynchronous processes in Section 5.2. Finally, in Section 5.3, we show that partial round abstraction is compositional on causal and asynchronous processes.

5.1 Justified Traces and Causal Processes

5.1.1 Signatures

We revisit signatures to add a causality relation between labels.

Definition 5.1 (Signature). *A signature A is a finite set equipped with a labelling function and a causality relation. Formally, it is a triple $\langle L_A, \pi_A, \vdash_A \rangle$ where,*

- L_A *is a finite* set of labels.
- $\pi_A : L_A \to \{i, o\}$ *maps each label to an* input/output *polarity.*
- $\vdash_A \subseteq (L_A + \{\star\}) \times L_A$ *is a relation called* causality *that satisfies the following conditions.*
  - *Let $\vdash_A^T$ be the transitive closure of $\vdash_A$. For all $a, b \in L_A$, if $a \vdash_A^T b$, then $b \not\vdash_A^T a$.*
  - *If $\star \vdash_A a$, then $\pi_A(a) = i$ and $[b \vdash_A a \Leftrightarrow b = \star]$.*
  - *If $a \vdash_A b$ and $a \neq \star$, then $\pi_A(a) \neq \pi_A(b)$.*

Signatures are similar to game semantic *arenas* —in particular, to their formulation in [AM99a].

The causality relation, akin to game semantic *enabling*, will turn out to be technically important. Intuitively, it models the fact that a b-event cannot happen unless some a-event causes it. Note that causality is both descriptive, when an input causes an output, but also prescriptive, when an output can require the environment to only produce certain inputs. The asymmetry condition on the causality relation indicates that it must be acyclic. This allows us to rule out causality cycles in rounds. We denote by I_A the elements of L_A caused by the special label $\star$ and call them *initial*.

We modify the *tensor* and *arrow* signatures to take causality into account. Intuitively, the former amounts to grouping two signatures together, while the latter corresponds to forming a function space, where one signature can be queried by the other. They are defined as follows.

$$\begin{aligned}
L_{A\otimes B} &= L_A + L_B \\
\pi_{A\otimes B} &= [\pi_A, \pi_B] \\
\star \vdash_{A\otimes B} a &\Leftrightarrow \star \vdash_A a \text{ or } \star \vdash_B a \\
a \vdash_{A\otimes B} b &\Leftrightarrow a \vdash_A b \text{ or } a \vdash_B b
\end{aligned}$$

$$\begin{aligned}
L_{A\Rightarrow B} &= L_A + L_B \\
\pi_{A\Rightarrow B} &= [\pi_A^*, \pi_B] \\
\star \vdash_{A\Rightarrow B} a &\Leftrightarrow \star \vdash_B a \\
a \vdash_{A\Rightarrow B} b &\Leftrightarrow a \vdash_A b \text{ or } a \vdash_B b \text{ or } (\star \vdash_B a \text{ and } \star \vdash_A b)
\end{aligned}$$

Again, note the similarity of these definitions to those of the *product* and *exponential* arenas in Game Semantics [HO00, AJM00], in particular, those in [AM99a].

5.1.2 Traces

We now modify the definition of pre-trace to include *pointers*. The causality relation at the level of signatures allows for an event to have multiple *a priori* causers. However, traces record 'occurrences', each of which can only have a single cause.

Definition 5.2 (Partially justified pre-trace)**.** *A* partially justified locally synchronous pre-trace *over a signature A is a quadruple $\langle E, \preceq, \lambda, \curvearrowleft \rangle$ where E is a finite set of events, $\preceq$ is a total preorder on E, $\lambda : E \to L_A$ is a function mapping events to labels in A and $\curvearrowleft : E \setminus \{e \in E \mid \lambda(e) \in I_A\} \rightharpoonup E$ is a partial function such that $\curvearrowleft(e) = e'$ implies $e' \preceq e$ and $\lambda(e') \vdash_A \lambda(e)$.*

If $\curvearrowleft(e) = e'$, we write $e' \curvearrowleft e$ and say that e' causes or justifies e. We denote by $\Delta_J(A)$ the set of partially justified pre-traces over A.

Justified traces are pre-traces where the justification function is total.

Definition 5.3 (Justified synchronous trace)**.** *A* justified locally synchronous trace *is a justified pre-trace $\langle E, \preceq, \lambda, \curvearrowleft \rangle$ with singular events and where $\curvearrowleft$ is a function.*

We denote by $\Theta_J(A)$ the set of justified traces over A. Trace equivalence is modified accordingly.

Definition 5.4 (Justified trace equivalence)**.** *Two justified pre-traces are considered equivalent, written $s \cong t$, if they only differ in the choice of their carrier sets. Formally, pre-traces $s = \langle E_s, \preceq_s, \lambda_s, \curvearrowleft_s \rangle$ and $t = \langle E_t, \preceq_t, \lambda_t, \curvearrowleft_t \rangle$ are equivalent if there exists a bijection $\phi : E_s \to E_t$ satisfying for all events $e_1, e_2 \in E_s$*

- $e_1 \preceq_s e_2 \Leftrightarrow \phi(e_1) \preceq_t \phi(e_2)$ *and*
- $\lambda_s = \lambda_t \circ \phi$ *and*
- $e_1 \curvearrowleft_s e_2 \Leftrightarrow \phi(e_1) \curvearrowleft_t \phi(e_2)$.

We will work with the quotient sets $\Delta_J/\cong$ and $\Theta_J/\cong$, only distinguishing justified pre-traces and justified traces up to $\cong$-equivalence. In the sequel, for justified traces s and t, we will write $s = t$ to mean that they belong to the same equivalence class.

5.1.3 Processes

We first need to refine our previous notion of prefix-closure. Let us define a function $\lceil - \rceil : \Theta_J(A) \to \Theta(\langle L_A, \pi_A \rangle)$ which maps each justified trace u to $\langle E_u, \preceq_u, \lambda_u \rangle$; that is, to u without its justification structure.

Definition 5.5 (Prefix)**.** *Let u be a justified trace over A such that $\lceil u \rceil = \lceil v \rceil \cdot w$. We say that v is a prefix of u when $\curvearrowleft_v = \curvearrowleft_u \cap (E_v \times E_v)$.*

We define the *prefix* preorder $\preccurlyeq$ on $\Theta_J(A)$ as the least reflexive and transitive relation satisfying for justified traces s and t, $s \preccurlyeq t$ if s is a prefix of t. The next lemma directly follows from the previous definition.

Lemma 5.6. *The prefix of a justified trace is a justified trace.*

We call processes consisting of justified traces *causal*.

Definition 5.7 (Causal process)**.** *A causal* process *σ over signature A, $\sigma : A$, is a nonempty and $\preccurlyeq$-downward closed set of justified traces.*

Let $s \upharpoonright A$ be the pre-trace obtained from s by deleting all events with labels not belonging to L_A. For any signatures A, B and C we define the set of *interaction traces*, written $int_J(A, B, C)$, as $\Theta_J((A \Rightarrow B) \Rightarrow C)$. Given another signature D, let $int_J(A, B, C, D)$ be the set $\Theta_J((A \Rightarrow B) \Rightarrow C) \Rightarrow D)$.

Definition 5.8 (Interaction)**.** *Let $\sigma : A \to B$ and $\tau : B \to C$ be two causal processes. Their interaction is $\sigma \between \tau = \{u \in int_J(A, B, C) \mid u \upharpoonright A + B \in \sigma \text{ and } u \upharpoonright B + C \in \tau\}$.*

Let $u \in int_J(A, B, C)$ be an interaction trace. We define $u \upharpoonright A + C$ as the subtrace of u consisting of all events labelled by A and C. Moreover, for all $a, b, c \in E_u$ such that $\lambda_u(a) \in L_A, \lambda_u(b) \in L_B, \lambda_u(c) \in L_C$, if $c \curvearrowleft_u b$ and $b \curvearrowleft_u a$, then $c \curvearrowleft_{u \upharpoonright A+C} a$.

Definition 5.9 (Composition)**.** *Let $\sigma : A \to B$ and $\tau : B \to C$ be two causal processes. Their composition is $\sigma; \tau : A \to C = \{u \upharpoonright A + C \mid u \in \sigma \between \tau\}$.*

The result below indicates the formalism so far makes sense, allowing us to model the function space and function application.

Theorem 5.10. *Causal processes form a closed symmetric monoidal category, which we call* $\mathbf{SynProc}_P$.

The proof is detailed in the next section.

5.1.4 A Category of Synchronous Causal Processes

We need a new definition for the identity morphism, which now takes pointers into account.

Definition 5.11 (Identity morphism)**.** *The identity morphism for object A, denoted by id_A, is the causal process consisting of traces u over $A_1 \Rightarrow A_2$ satisfying, for all events e in E_u, there exists an event $e' \neq e$, such that*

1. $e \approx_u e'$ *and*
2. $[\lambda_u(e) = inl(a)$ *if and only if* $\lambda_u(e') = inr(a)]$ *and* $[\lambda_u(e) = inr(a)$ *if and only if* $\lambda_u(e') = inl(a)]$ *and*
3. (a) *if* $\lambda_u(e) \in I_{A_1}$*, then* $e' \curvearrowleft_u e$
   (b) *if* $\lambda_u(e) \in I_{A_2}$*, then* $e \curvearrowleft_u e'$
   (c) *if* $\lambda_u(e) \notin I_{A_1}$ *and* $\lambda_u(e) \notin I_{A_2}$*, then there are* $e_1, e_2 \in E_u$ *such that* $e_1 \curvearrowleft_u e$ *and* $e_2 \curvearrowleft_u e'$ *and* $e_1 \approx_u e_2$ *and* $[\lambda_u(e_1) = inl(a')$ *if and only if* $\lambda_u(e_2) = inr(a')]$ *and* $[\lambda_u(e_1) = inr(a')$ *if and only if* $\lambda_u(e_2) = inl(a')]$

where $a, a' \in L_A$. For any two $e, e' \in E_u$, if conditions 1–3 hold, we write $e \leftrightarrows_u e'$.

The tensor product and the evaluation morphism are as defined as in Chapter 3 but we require their traces to be justified. Trace projection is modified as follows.

Definition 5.12 (Justified trace projection)**.** *Projection on justified traces is a function* $out : \Theta_J(A + B) \to \Delta_J(A)$ *such that for every* $t = \langle E, \preceq, \lambda, \curvearrowleft \rangle$ *in* $\Theta_J(A + B)$

$$out(t) = \langle E', \preceq', \lambda' \rangle$$

where

- $E' = \{e \in E \mid \lambda(e) \in lin(L_A)\}$,

- $\preceq' = \preceq \cap (E' \times E')$,
- $\lambda' = lout \circ \lambda$ *where lout is the projection on labels lout* : $L_A + L_B \rightharpoonup L_A$.
- $\curvearrowleft = \curvearrowleft \cap (E' \times E')$

We begin by showing that $\mathbf{SynProc}_P$ is a category. In the following proofs, we sometimes abuse notation by omitting label injections when no confusion may arise.

First, note that projection and composition preserve singularity. The proofs are the exactly the same as for Lemma 3.20 and Lemma 3.21.

Next, we show that composition is well-defined.

Lemma 5.13. *The composition of causal processes consists of justified traces.*

Proof. Let $\sigma : A \to B$ and $\tau : B \to C$ be causal processes and $v \in \sigma;\tau$. Take $u \in \sigma \between \tau$ such that $u \upharpoonright A + C = v$. Since u is an interaction of justified traces, it is itself justified. We have to show that for any event $e \in E_v$ such that $\lambda_v(e) \notin I_C$, there is $e' \in E_v$ such that $e' \preceq_v e$ and $\lambda_v(e') \vdash \lambda_v(e)$ and $e' \curvearrowleft_v e$.

1. If $\lambda_v(e) \in L_C$, then $\lambda_u(e) \in L_C$. Therefore, there is a C-event $e' \in E_u$ such that $e' \preceq_u e$ and $\lambda_u(e') \vdash \lambda_u(e)$ and $e' \curvearrowleft_u e$. After projection over $A + C$, e' justifies e.
2. If $\lambda_v(e) \in L_A$, then $\lambda_u(e) \in L_A$. There are two cases: either e is caused by an A-event in u or e is caused by a B-event in u.
   (a) In the former case, we use the same reasoning as Case 1 to conclude that there is an A-event in v that justifies e.
   (b) In the latter case, e must be initial in A. Its cause in $u \upharpoonright A + B$ must be an initial event e' in B. So, e' must be justified in $u \upharpoonright B + C$ by an initial event e'' in C. By the definition of composition, e'' justifies e in v. ■

The composition of causal processes preserves prefix-closure.

Lemma 5.14. *Let s be a trace over* $A + B$. *We have* $\lceil s \rceil \upharpoonright B = \lceil s \upharpoonright B \rceil$.

Proof. By inspection of the definition of trace projection and $\lceil - \rceil$. ■

Lemma 5.15. *Composition preserves prefix-closure.*

Proof. Let $\sigma : A \to B$ and $\tau : B \to C$ be processes. We begin by showing that $\sigma \lightning \tau$ is prefix-closed. Take $u \in \sigma \lightning \tau$; that is, $u \upharpoonright A + B = s \in \sigma$ and $u \upharpoonright B + C = t \in \tau$. Let $\lceil u \rceil = \lceil v \rceil \cdot w$ such that v is a prefix of u, i.e. $\curvearrowleft_v = \curvearrowleft_u \cap (E_v \times E_v)$.

Since $\lceil u \rceil = \lceil v \rceil \cdot w$, we have $\lceil u \rceil \upharpoonright A + B = \lceil v \rceil \cdot w \upharpoonright A + B$. Using Lemma 3.22, we obtain $\lceil u \rceil \upharpoonright A + B = (\lceil v \rceil \upharpoonright A + B) \cdot (w \upharpoonright A + B)$. By Lemma 5.14, we get $\lceil u \upharpoonright A + B \rceil = (\lceil v \upharpoonright A + B \rceil) \cdot (w \upharpoonright A + B)$. Moreover, from $\curvearrowleft_v = \curvearrowleft_u \cap (E_v \times E_v)$ we get $\curvearrowleft_{v \upharpoonright A+B} = \curvearrowleft_{u \upharpoonright A+B} \cap (E_{v \upharpoonright A+B} \times E_{v \upharpoonright A+B})$ by definition of trace projection. We deduce that $v \upharpoonright A + B$ is a prefix of $u \upharpoonright A + B$. Since σ is prefix-closed and $u \upharpoonright A + B \in \sigma$, it follows that $v \upharpoonright A + B \in \sigma$.

We use the same reasoning to conclude that $v \upharpoonright B + C \in \tau$ and therefore $v \in \sigma \lightning \tau$. So, $\sigma \lightning \tau$ is prefix-closed.

Next, we show that $\sigma ; \tau$ is prefix-closed. Take $v \in \sigma ; \tau$ and let $\lceil v \rceil = \lceil v_1 \rceil \cdot v_2$ such that v_1 is a prefix of v, i.e. $\curvearrowleft_{v_1} = \curvearrowleft_v \cap (E_{v_1} \times E_{v_1})$. By Definition 5.9, we know that there exists $u \in \sigma \lightning \tau$ such that $u \upharpoonright A + C = v$. So, $\lceil u \upharpoonright A + C \rceil = \lceil v_1 \rceil \cdot v_2$ and $\curvearrowleft_{v_1} = \curvearrowleft_{u \upharpoonright A+C} \cap (E_{v_1} \times E_{v_1})$.

Without loss of generality, let $E_v = E_{u \upharpoonright A+C}$. We show that u has a prefix u_1, i.e. $\lceil u \rceil = \lceil u_1 \rceil \cdot u_2$ and $\curvearrowleft_{u_1} = \curvearrowleft_u \times (E_{u_1} \times E_{u_1})$, satisfying $u_1 \upharpoonright A + C = v_1$. Let u_1 be defined as follows.

- $E_{u_1} = E_{v_1} + \{e \in E_u \mid \lambda_u(e) \in L_B \text{ and } (\forall e' \in last(v_1))(e \preceq_u e')\}$
- $\preceq_{u_1} = \preceq_u \cap (E_{u_1} \times E_{u_1})$
- $\lambda_{u_1} = \lambda_u \upharpoonright E_{u_1}$
- $\curvearrowleft_{u_1} = \curvearrowleft_u \cap (E_{u_1} \times E_{u_1})$

Let u_2 be defined as follows.

- $E_{u_2} = E_{v_2} + \{e \in E_u \mid \lambda_u(e) \in L_B \text{ and } (\forall e' \in last(v_1))(e' \preceq_u e \text{ and } e \npreceq_u e')\}$
- $\preceq_{u_2} = \preceq_u \cap (E_{u_2} \times E_{u_2})$
- $\lambda_{u_2} = \lambda_u \upharpoonright E_{u_2}$

It is clear, by construction, that $u_1 \upharpoonright A + C = v$. Since $\sigma \between \tau$ is prefix-closed, $u_1 \in \sigma \between \tau$ and hence $v_1 \in \sigma; \tau$. ∎

Now, we can prove that the basic axioms of a category hold. From its definition, the identity morphism is a causal process.

Lemma 5.16. *For each object A in $\mathbf{SynProc}_P$, id_A is the identity morphism.*

Proof. The proof is exactly the same as for Lemma 3.24 but we need to additionally show that the pointer structure is preserved.

Let $\phi : E_s \to E_{u \upharpoonright A+B'}$ be defined as in the first part of the proof of Lemma 3.24. We need to prove that $(\forall e, e' \in E_s)(e \curvearrowleft_s e' \Leftrightarrow \phi(e) \curvearrowleft_{u \upharpoonright A+B'} \phi(e'))$.

Without loss of generality, let $E_{u \upharpoonright A+B} = E_s$ and $E_{u \upharpoonright B'} = E_{t \upharpoonright B'}$. First, we show the left to right implication. Let $e \curvearrowleft_s e'$ which implies $e \preceq_s e'$ and $\lambda_s(e) \vdash_{A \Rightarrow B} \lambda_s(e')$. We have already shown that $e \preceq_s e'$ iff $\phi(e) \preceq_{u \upharpoonright A+B'} \phi(e')$. It is impossible for A-events to justify B-events, so we have the following three cases.

1. If $\lambda_s(e) \in inl(L_A)$ and $\lambda_s(e') \in inl(L_A)$, then
   - $\phi(e) = e$ and $\phi(e') = e'$
   - $e \curvearrowleft_u e' \therefore \phi(e) \curvearrowleft_{u \upharpoonright A+B'} \phi(e')$
2. If $\lambda_s(e) \in inr(L_B)$ and $\lambda_s(e') \in inl(L_A)$, then
   - $\phi(e) \approx_u e$ and $\phi(e') = e'$ such that $\lambda_u(e) = inl(inr(b))$ implies $\lambda_u(\phi(e)) = inr(b)$, $b \in L_B$
   - $\phi(e) \curvearrowleft_{u \upharpoonright B+B'} e$ because e is initial in B (using the definition of identity)
   - $\phi(e) \curvearrowleft_{u \upharpoonright B+B'} e$ and $e \curvearrowleft_s e'$ implies $\phi(e) \curvearrowleft_{u \upharpoonright A+B'} \phi(e')$
3. If $\lambda_s(e) \in inr(L_B)$ and $\lambda_s(e') \in inr(L_B)$, then
   - $\phi(e) \approx_u e$ and $\phi(e') \approx_u e'$ such that $[\lambda_u(e) = inl(inr(b))$ implies $\lambda_u(\phi(e)) = inr(b)]$ and $[\lambda_u(e') = inl(inr(b'))$ implies $\lambda_u(\phi(e')) = inr(b')]$, $b, b' \in L_B$
   - Using the definition of identity, it follows that $\phi(e) \curvearrowleft_{u \upharpoonright B+B'} \phi(e')$.

- Since $\lambda_{u\restriction B+B'}(\phi(e)), \lambda_{u\restriction B+B'}(\phi(e)) \in L_{B'}$, we have $\phi(e) \curvearrowleft_u \phi(e')$ which implies $\phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e')$

The proof of the left to right implication is similar using ϕ^{-1}.

In the second part of the proof, let the pointer structure of u be defined as follows. For all $e, e' \in E_s$,

J1 If $e, e' \in E_{s\restriction A}$, then $e \curvearrowleft_s e' \Leftrightarrow in_1(e) \curvearrowleft_u in_1(e')$.

J2 If $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $e \curvearrowleft_s e' \Leftrightarrow in_2(e) \curvearrowleft_u in_1(e')$.

J3 If $e, e' \in E_{s\restriction B}$, then $e \curvearrowleft_s e' \Leftrightarrow in_2(e) \curvearrowleft_u in_2(e') \Leftrightarrow in_3(e) \curvearrowleft_u in_3(e')$.

J4 $e \in E_{s\restriction B}$ and $\lambda_s(e) \in I_B \Leftrightarrow in_3(e) \curvearrowleft_u in_2(e)$.

J5 The only pointers in $\curvearrowleft_u$ are those specified by (J1)–(J4); that is,

- if $e \in E_{s\restriction A}$ and $e' \in E_{s\restriction B}$, then $[in_1(e) \not\curvearrowleft_u in_2(e')$ and $in_1(e) \not\curvearrowleft_u in_3(e')]$,
- if $e \in E_{s\restriction B}$, then $in_2(e) \not\curvearrowleft_u in_3(e')$,
- if $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $in_3(e) \not\curvearrowleft_u in_1(e')$.

By construction, u is justified because all A-events and B-events are justified according to s whereas B'-events are justified according to the definition of identity. ■

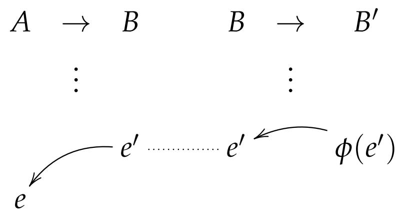


Figure 5.1: Lemma 5.16, Case 2

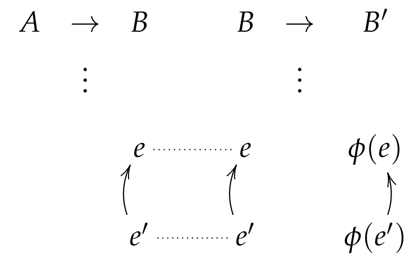


Figure 5.2: Lemma 5.16, Case 3

Lemma 5.17. *Let s be a justified interaction trace over $int_J(A, B, C)$ and t be a justified trace over $C \to D$. We have $(s \restriction_B^{A+C}) \lightning t = (s \lightning t) \restriction_B^{A+C+D}$.*

Proof. First, we prove $(s \upharpoonright_B^{A+C}) \between t \subseteq (s \between t) \upharpoonright_B^{A+C+D}$. Let $u \in (s \upharpoonright_B^{A+C}) \between t$; that is, $u \in int_J(A, C, D)$ and $u \upharpoonright A + C = s \upharpoonright A + C$ and $u \upharpoonright C + D = t$. To show that $u \in (s \between t) \upharpoonright_B^{A+C+D}$, we construct a trace $v \in int_J(A, B, C, D)$ such that $v \upharpoonright A + C + D = u$ and $v \upharpoonright A + B + C = s$ and $v \upharpoonright C + D = t$ as follows.

Without loss of generality, we assume that $E_{s \upharpoonright A+C} = E_{u \upharpoonright A+C}$. This allows us to simplify the proof by avoiding trace equivalence maps.

We extend the definition of the trace v from the proof of Lemma 3.26 by adding justification pointers.

- $E_v = E_{u \upharpoonright A} + E_{s \upharpoonright B} + E_{u \upharpoonright C} + E_{u \upharpoonright D}$. In the sequel, we will omit injections on events; for example, we will write e where it should be $in_j(e), j \in \{1, 2, 3, 4\}$. This is possible because the sets $E_{u \upharpoonright A}, E_{s \upharpoonright B}, E_{u \upharpoonright C}$ and $E_{u \upharpoonright D}$ are disjoint. It should improve readability at the cost of abusing notation.
- $\lambda_v = \lambda_{u \upharpoonright A} + \lambda_{s \upharpoonright B} + \lambda_{u \upharpoonright C} + \lambda_{u \upharpoonright D}$
- We define $\preceq_v$ in two steps. Let $\preceq_1$ be the relation satisfying the following.
  - $(\forall e, e' \in E_{u \upharpoonright A} + E_{u \upharpoonright C} + E_{u \upharpoonright D})(e \preceq_u e' \Leftrightarrow e \preceq_1 e')$
  - $(\forall e, e' \in E_{u \upharpoonright A} + E_{s \upharpoonright B} + E_{u \upharpoonright C})(e \preceq_s e' \Leftrightarrow e \preceq_1 e')$

  Note that the only events that are not comparable using $\preceq_1$ are B-events with respect to D-events. We define a second relation $\preceq_2$ as follow. For all $e \in E_{s \upharpoonright B}$ and $e' \in E_{u \upharpoonright D}$

  - $(\exists e'' \in E_v)(e' \preceq_1 e''$ and $e'' \preceq_1 e)$ if and only if $e' \preceq_2 e$
  - $(\exists e'' \in E_v)(e \preceq_1 e''$ and $e'' \preceq_1 e')$ or $(\forall e'' \in E_v)\big((e \not\preceq_1 e''$ or $e'' \not\preceq_1 e')$ and $(e' \not\preceq_1 e''$ or $e'' \not\preceq_1 e)\big)$ if and only if $e \preceq_2 e'$

  We then set $\preceq_v$ to be the union of $\preceq_1$ and $\preceq_2$.
- We define $\curvearrowleft_v$ as follows. For all $e, e' \in E_v$,

  J1 if $e, e' \in E_{s \upharpoonright A}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e' \Leftrightarrow e \curvearrowleft_u e'$,

  J2 if $e, e' \in E_{s \upharpoonright B}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e$,

  J3 if $e, e' \in E_{s \upharpoonright C}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e' \Leftrightarrow e \curvearrowleft_u e'$,

J4 if $e, e' \in E_{s \upharpoonright D}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_u e'$,

J5 if $e \in E_{s \upharpoonright B}$ and $e' \in E_{s \upharpoonright A}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e'$,

J6 if $e \in E_{s \upharpoonright C}$ and $e' \in E_{s \upharpoonright B}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e'$,

J7 if $e \in E_{u \upharpoonright D}$ and $e' \in E_{u \upharpoonright C}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_u e'$.

J8 The only pointers in $\curvearrowleft_v$ are those specified by (J1)–(J7); that is, if $[e \in E_{u \upharpoonright D}$ and $e' \in E_{s \upharpoonright B}]$ or $[e \in E_{u \upharpoonright D}$ and $e' \in E_{s \upharpoonright A}]$ or $[e \in E_{s \upharpoonright C}$ and $e' \in E_{u \upharpoonright D}]$ or $[e \in E_{s \upharpoonright C}$ and $e' \in E_{s \upharpoonright A}]$ or $[e \in E_{s \upharpoonright B}$ and $e' \in E_{u \upharpoonright D}]$ or $[e \in E_{s \upharpoonright B}$ and $e' \in E_{s \upharpoonright C}]$ or $[e \in E_{s \upharpoonright A}$ and $e' \in E_{u \upharpoonright D}]$ or $[e \in E_{s \upharpoonright A}$ and $e' \in E_{s \upharpoonright C}]$ or $[e \in E_{s \upharpoonright A}$ and $e' \in E_{s \upharpoonright B}]$, then $e \not\curvearrowleft_v e'$.

We have shown in the proof of Lemma 3.26 that $\preceq_v$ is a total preorder and that v has singular events. We additionally need to show that v is a justified. This follows from the definition of $\curvearrowleft_v$ and the fact that u and s are justified.

Now we show that the conditions set above are satisfied by $\preceq_v$.

We first prove that $v \upharpoonright A + B + C = s$. We have shown in the proof of Lemma 3.26 that $\lceil v \upharpoonright A + B + C \rceil = \lceil s \rceil$, i.e. $\langle E_s, \preceq_s, \lambda_s \rangle = \langle E_{v \upharpoonright A+B+C}, \preceq_{v \upharpoonright A+B+C}, \lambda_{v \upharpoonright A+B+C} \rangle$. There is a bijection $\phi : E_s \to E_{v \upharpoonright A+B+C}$ defined as

$$\phi(e) = \begin{cases} in_1(e) & \text{if } \lambda_s(e) \in inl(inl(L_A)) \\ in_2(e) & \text{if } \lambda_s(e) \in inl(inr(L_B)) \\ in_3(e) & \text{if } \lambda_s(e) \in inr(L_C) \end{cases}$$

We need to show that $(\forall e, e' \in E_s)(e \curvearrowleft_s e' \Leftrightarrow \phi(e) \curvearrowleft_{v \upharpoonright A+B+C} \phi(e'))$. This follows from the definition of $\curvearrowleft_v$ and the fact that $s \upharpoonright A + C = u \upharpoonright A + C$.

The proof that $v \upharpoonright A + C + D = u$ is similar.

The proof that $v \upharpoonright C + D = t$ follows from the following two facts: $v \upharpoonright A + C + D = u$ and $u \upharpoonright C + D = t$. Now, we prove $(s \upharpoonright_B^{A+C}) \lightning t \supseteq (s \lightning t) \upharpoonright_B^{A+C+D}$. Let $u \in s \lightning t$; that is, $u \in int_J(A, B, C, D)$ and $u \upharpoonright A + B + C = s$ and $u \upharpoonright C + D = t$. Applying the projection over $A + C + D$, we get $u \upharpoonright A + C + D \in int_J(A, C, D)$ and $u \upharpoonright A + C = s \upharpoonright A + C$ and $u \upharpoonright C + D = t$. So, $u \in (s \upharpoonright_B^{A+C}) \lightning t$. ∎

Corollary 5.18. *Let $\sigma : A \otimes B \to C$ and $\tau : C \to D$ be causal processes. We have* $(\sigma \upharpoonright^{A+C}_{B}) \lightning \tau = (\sigma \lightning \tau) \upharpoonright^{A+C+D}_{B}$.

Lemma 5.19. *Let s be an interaction trace over $int_J(A, B, C, D)$. We have* $(s \upharpoonright^{A+C+D}_{B}) \upharpoonright^{A+D}_{C} = (s \upharpoonright^{A+B+D}_{C}) \upharpoonright^{A+D}_{B} = s \upharpoonright^{A+D}_{B+C}$.

Proof. By inspection of the definition of composition. ■

After projecting interaction traces, the pointer structure is preserved: so for events $a, b, c, d \in E_u$ where $u \in \sigma \lightning \tau \lightning \gamma$ and pointers assigned following Figure 5.3, whether we remove B-events first, then C-events, or remove C-events first then B-events second, we will have $d \curvearrowleft a$.

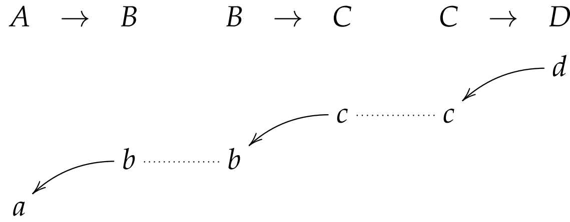


Figure 5.3: Pointer structure inheritance in Lemma 5.20

Lemma 5.20. *Composition is associative.*

Proof. Let $\sigma : A \to B$, $\tau : B \to C$ and $\gamma : C \to D$ be processes. It is easy to see, following Lemma 3.29, that $(\sigma \lightning \tau) \lightning \gamma = \sigma \lightning (\tau \lightning \gamma)$. Let us refer to this result as ($\bigstar$).

By Definition 5.9, composition consists in hiding the internal channels in interaction traces and extending justification pointers. So, $\sigma; \tau = \sigma \lightning \tau \upharpoonright^{A+C}_{B}$.

$$
\begin{aligned}
(\sigma;\tau);\gamma &= ((\sigma \lightning \tau \upharpoonright^{A+C}_{B}) \lightning \gamma) \upharpoonright^{A+D}_{C} \\
&= ((\sigma \lightning \tau) \lightning \gamma \upharpoonright^{A+C+D}_{B}) \upharpoonright^{A+D}_{C} \text{ by Corollary 5.18} \\
&= (\sigma \lightning \tau) \lightning \gamma \upharpoonright^{A+D}_{B+C} \text{ by Lemma 5.19} \\
&= \sigma \lightning (\tau \lightning \gamma) \upharpoonright^{A+D}_{B+C} \text{ by } (\bigstar) \\
&= (\sigma \lightning (\tau \lightning \gamma) \upharpoonright^{A+B+D}_{C}) \upharpoonright^{A+D}_{B} \text{ by Lemma 5.19} \\
&= (\sigma \lightning (\tau \lightning \gamma \upharpoonright^{B+D}_{C})) \upharpoonright^{A+D}_{B} \text{ by Corollary 5.18} \\
&= \sigma; (\tau; \gamma).
\end{aligned}
$$
■

The following Proposition sums up our results so far.

Proposition 5.21. $\mathbf{SynProc}_P$ *is a category.*

Next, we demonstrate that $\mathbf{SynProc}_P$ is a monoidal category.

Lemma 5.22. *The tensor product of causal processes consists of justified traces.*

Proof. This follows directly from the definitions of justified trace and tensor product. ∎

We verify that the tensor is a functor.

Lemma 5.23. *The tensor product preserves the identity morphism.*

Proof. We want to show that $id_A \otimes id_B = id_{A \otimes B}$.

First, we prove $id_A \otimes id_B \subseteq id_{A\otimes B}$. Let $u \in id_A \otimes id_B$; that is, $u \in \Theta_J(A \otimes B \Rightarrow A' \otimes B')$ and $u \upharpoonright A + A' \in id_A$ and $u \upharpoonright B + B' \in id_B$. Using Definition 5.11, we get $u \in \Theta_J(A \otimes B \Rightarrow A' \otimes B')$ and $(\forall e \in E_{u\upharpoonright A+A'}, \exists e' \in E_{u\upharpoonright A+A'})(e \leftrightarrows_{u\upharpoonright A+A'} e')$ and $(\forall e \in E_{u\upharpoonright B+B'}, \exists e' \in E_{u\upharpoonright B+B'})(e \leftrightarrows_{u\upharpoonright B+B'} e')$. Since $E_u = E_{u\upharpoonright A+A'} \cup E_{u\upharpoonright B+B'}$ and $(\forall e, e' \in E_u)$(if $e \approx_{u\upharpoonright A+A'} e'$ or $e \approx_{u\upharpoonright B+B'} e'$, then $e \approx_{u'} e'$) and for all $a \in E_{u\upharpoonright A+A'}, b \in E_{u\upharpoonright B+B'}$, we have $\lambda_{u\upharpoonright A+A'}(a) = \lambda_u(a)$ and $\lambda_{u\upharpoonright B+B'}(b) = \lambda_u(b)$, we conclude that for all $e \in E_u$, there is $e' \in E_u$ such that $e \leftrightarrows_u e'$. Therefore, $u \in id_{A\otimes B}$.

Now, we prove $id_A \otimes id_B \supseteq id_{A\otimes B}$. Let $u \in id_{A\otimes B}$; that is, $u \in \Theta_J(A \otimes B \Rightarrow A' \otimes B')$ and $(\forall e \in E_u, \exists e' \in E_u)(e \leftrightarrows_u e')$. After projecting u over $A + A'$, we get $u \upharpoonright A + A' \in \Theta_J(A \Rightarrow A')$ and $(\forall e \in E_{u\upharpoonright A+A'}, \exists e' \in E_{u\upharpoonright A+A'})(e \leftrightarrows_{u\upharpoonright A+A'} e')$; that is, $u \upharpoonright A + A' \in id_A$. Projecting u over $B + B'$ yields $u \upharpoonright B + B' \in \Theta_J(B \Rightarrow B')$ and $(\forall e \in E_{u\upharpoonright B+B'}, \exists e' \in E_{u\upharpoonright B+B'})(e \leftrightarrows_{u\upharpoonright B+B'} e')$; that is, $u \upharpoonright B + B' \in id_B$. So, $u \in id_A \otimes id_B$. ∎

Note that, in signature $A \otimes B \Rightarrow C \otimes D$, initial D-events may cause initial A-events. However, in $id_{A\otimes B}$, the definition of identity assigns pointers in a way that guarantees only initial B'-events cause initial B-events. This is depicted in Figure 5.4.

Lemma 5.24. *Let* $\sigma : A \to B$, $\gamma : B \to C$ *and* $\tau : D \to E$ *be causal processes. We have* $(\sigma \lightning \gamma \upharpoonright_B^{A+C}) \otimes \tau = ((\sigma \lightning \gamma) \otimes \tau) \upharpoonright_B^{A+C+D+E}$.

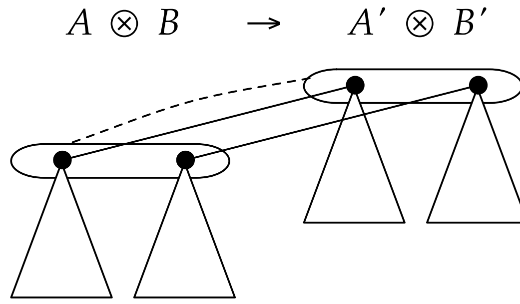


Figure 5.4: Causality in $id_{A \otimes B}$.

Proof. We use the same strategy underlying the proof of Lemma 5.17. ■

Lemma 5.25. *The tensor preserves composition.*

Proof. The proof is the same as for Lemma 3.34 but we use Lemma 5.24 instead of Lemma 3.35. ■

Next, we show the existence of the monoidal natural isomorphisms. The proofs of naturality are omitted.

Lemma 5.26. *The tensor product is associative. That is, there exists a natural isomorphism called the* associator, *assigning to each triple of objects A, B, C an isomorphism,*

$$\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$$

such that the pentagon diagram in Figure 3.1 commutes.

Proof. It is clear from the definition of tensor product that $(A \otimes B) \otimes C$ and $A \otimes (B \otimes C)$ have the same labels tagged differently. Therefore, there exists an isomorphism $(A \otimes B) \otimes C \to A \otimes (B \otimes C)$ for every triple of objects A, B, C, whose action is to retag the labels and whose traces are equivalent, up to retagging of labels by *inl* and *inr*, to those in $id_{A \otimes B \otimes C}$. We call this isomorphism $\alpha_{A,B,C}$. The coherence condition in Figure 3.1 then follows from the definitions of the tensor product and α. ■

Lemma 5.27. *The tensor product has I as left and right identity. That is, there exists two natural isomorphisms called* left and right unitors, *assigning to each object A the following isomorphisms,*

$$\lambda_A : I \otimes A \to A$$
$$\rho_A : A \otimes I \to A$$

such that the triangle diagram in Figure 3.2 commutes.

Proof. Since the tensor product is (roughly) defined as the disjoint union on objects, and I is the object with an empty label set, $\langle \emptyset, \pi_\emptyset \rangle$, it follows that for any $A \in \mathbf{SynProc}_P$, $I \otimes A$ and $A \otimes I$ are clearly isomorphic to A as tensoring with I only retags its elements. Therefore, we define the left identity isomorphism λ_A and the right identity isomorphism ρ_A as those with the signatures above, and which consist of exactly the same traces, up to retagging of labels by *inl* and *inr*, as id_A. The coherence condition in Figure 3.2 directly follows from the definitions of the tensor product, λ, and ρ. ■

Next, we show that $\mathbf{SynProc}_P$ is symmetric.

Proposition 5.28. $\mathbf{SynProc}_P$ *is a symmetric monoidal category. That is, there exists a natural isomorphism called* symmetry *that assigns to every pair of objects A, B an isomorphism, $\gamma_{A,B} : A \otimes B \to B \otimes A$ such that the hexagon diagrams in Figure 3.3 commute and $\gamma_{A,B}; \gamma_{B,A} = id_{A \otimes B}$.*

Proof. For any A, B in $\mathbf{SynProc}_P$, let $\gamma_{A,B} : A \otimes B \to B \otimes A$ be the isomorphism consisting of exactly the same traces, up to retagging of labels by *inl* and *inr*, as $id_{A \otimes B}$. It satisfies the coherence conditions in Figure 3.3. It is then clear that $\gamma_{A,B}; \gamma_{B,A}$ has the same traces, up to retagging of labels by *inl* and *inr*, as $id_{A \otimes B}; id_{A \otimes B}$. However, as $\gamma_{A,B}$ changes the tags of labels and $\gamma_{B,A}$ changes them back according to ($\star$), $\gamma_{A,B}; \gamma_{B,A} = id_{A \otimes B}; id_{A \otimes B} = id_{A \otimes B}$.

$$\begin{aligned}
L_{A \otimes B \Rightarrow A' \otimes B'} &= inl(inl(A)) \cup inl(inr(B)) \cup inr(inl(A')) \cup inr(inr(B')) \\
L_{A \otimes B \Rightarrow B' \otimes A'} &= inl(inl(A)) \cup inl(inr(B)) \cup inr(inl(B')) \cup inr(inr(A')) \quad (\star) \\
L_{B' \otimes A' \Rightarrow A \otimes B} &= inl(inl(B')) \cup inl(inr(A')) \cup inr(inl(A)) \cup inr(inr(B))
\end{aligned}$$

■

Finally, we show that $\mathbf{SynProc}_P$ is closed.

Proposition 5.29. $\mathbf{SynProc}_P$ *with* $- \otimes -$, I, $eval_{A,B}$ *is a closed symmetric monoidal category.*

Proof. We need to show that *eval* satisfies the universal property: for every morphism $\sigma : A \otimes B \to C$, there exists a unique morphism $\tau : A \to B \Rightarrow C$ such that $\sigma = (\tau \otimes id_B); eval_{B,C}$. It can be easily verified that τ with the same set of traces as σ satisfies the property. It is also clear that τ is unique: its traces have to be the same as those of σ since $eval_{B,C}$ is isomorphic to $id_B \otimes id_C$ and therefore, $(- \otimes id_B); eval_{B,C}$ only retags the events in the traces of any morphism placed in the hole $(-)$ according to the following.

$$L_{A \otimes B \Rightarrow C} = inl(inl(A)) \cup inl(inr(B)) \cup inr(C)$$
$$L_{A \Rightarrow B \Rightarrow C} = inl(A) \cup inr(inl(B)) \cup inr(inr(C))$$
■

5.2 Asynchronous Processes

By contrast to a synchronous trace, an asynchronous trace is one where the simultaneity relation is equal to the identity.

Definition 5.30 (Asynchronous trace). *A justified trace* $\langle E, \preceq, \lambda, \curvearrowleft \rangle$ *over signature* A *is* asynchronous *if* $\preceq$ *is a total order.*

Definition 5.30 reflects the failure of synchrony in asynchronous systems, as no two distinct events can be ascertained to occur precisely at the same time. We denote by $\Gamma(A)$ the set of asynchronous traces over A

For an asynchronous trace t of length n, we define a bijective enumeration $pos_t : \{1, \ldots, n\} \to E_t$ satisfying $(\forall e, e' \in E_t)(e \preceq_t e' \Leftrightarrow pos_t^{-1}(e) < pos_t^{-1}(e'))$.

As discussed in Section 2.2.2, traces of an asynchronous process must be closed under certain permutations of events, corresponding to inputs occurring earlier and outputs occurring later. To maintain consistency, we require that causality between events is not changed by the permutations. For this reason, asynchronous processes must determine the causal dependency between events.

Definition 5.31 (Saturation preorder). *We define a relation* $\lesssim^{sm}$ *on* $\Gamma(A)$ *satisfying* $s' \lesssim^{sm} s$ *when there is an event* $e \in E_s$ *and a bijection* $\phi : E_s \to E_{s'}$ *satisfying*

1. *For all* $e_1, e_2 \in E_s \setminus \{e\}$, *we have* $e_1 \preceq_s e_2$ *if and only if* $\phi(e_1) \preceq_{s'} \phi(e_2)$.

2. $\lambda_s = \lambda_{s'} \circ \phi$.

3. *For all* $e_1, e_2 \in E_s$, *we have* $e_1 \curvearrowleft_s e_2$ *if and only if* $\phi(e_1) \curvearrowleft_{s'} \phi(e_2)$.

4. (a) *e is an input and* $pos_{s'}^{-1}(\phi(e)) \leq pos_s^{-1}(e)$, *or*

   (b) *e is an output and* $pos_s^{-1}(e) \leq pos_{s'}^{-1}(\phi(e))$.

We define a preorder $\lesssim$ *on* $\Gamma(A)$ *as the least reflexive and transitive relation containing* $\lesssim^{sm}$.

This preorder is a reformulation of a saturation principle which has long been used to model asynchronous systems [Udd86, JJH90]. It has been used in game models for asynchronous languages [Lai01b, Lai05b, Lai06] and circuits [Fos07]. In particular, it is a formalisation of the definition that appears on [GM08].

Intuitively, the relation $\lesssim^{sm}$ relates a trace s to a trace s' when an input e occurs earlier in s' or an output e occurs later in s' and the justification pointers of s' are 'inherited' from s. It is useful to have a 'big-step' explicit version of the preorder.

Definition 5.32. *Let* $\lesssim^0$ *be the relation on* $\Gamma(A)$ *satisfying* $s' \lesssim^0 s$ *if there is a bijection* $\phi : E_s \to E_{s'}$ *such that the following conditions hold.*

1. $\lambda_s = \lambda_{s'} \circ \phi$

2. $(\forall e, e' \in E_s)(e \curvearrowleft_s e'$ *if and only if* $\phi(e) \curvearrowleft_s \phi(e'))$

3. *If* $e \preceq_s e'$ *and e is an input and* e' *is an output, then* $\phi(e) \preceq_{s'} \phi(e')$

We need to verify that the two definitions define the same preorder.

Lemma 5.33. *Let* $s, s' \in \Gamma(A)$. *We have* $s' \lesssim^0 s$ *if and only if* $s' \lesssim s$.

Proof. First, we show the left to right implication. Let $s' \lesssim^0 s$. Without loss of generality, we assume the bijection in Definition 5.32 is an identity, i.e. $E_s = E_{s'}$, $\lambda_s = \lambda_{s'}$ and $\curvearrowleft_s = \curvearrowleft_{s'}$. We show that if $|E_s| = n$, there exists a series of $n-1$ permutations of s such that $s' = s_n \lesssim s_{n-1} \lesssim \ldots \lesssim s_1 \lesssim s_0 = s$. We describe an algorithm that constructs s_{k+1} from s_k, $k \in \{0, \ldots, n-1\}$.

Define $B_1 = \emptyset$ and $B_{k+1} = \{e \in E_{s_k} \mid e \preceq_{s_k} pos_{s_k}(k)\}$. Define $A_{k+1} = \{e \in E_{s_k} \mid pos_{s_k}(k+1) \preceq_{s_k} e$ and $e \neq pos_{s'}(k+1)\}$.

We can now describe the algorithm.

- If $pos_{s_k}(k+1) = pos_{s'}(k+1)$, then $s_{k+1} = s_k$
- If $pos_{s_k}(k+1) \neq pos_{s'}(k+1)$, then s_{k+1} is defined as follows.
  - $E_{s_{k+1}} = E_{s_k}$
  - $\lambda_{s_{k+1}} = \lambda_{s_k}$
  - $\curvearrowleft_{s_{k+1}} = \curvearrowleft_{s_k}$
  - $\preceq_{s_{k+1}}$ is defined as follows, for all $e, e' \in E_{s_k}$.

    P1 If $e, e' \in B_{k+1}$, then $e \preceq_{s_k} e' \Leftrightarrow e \preceq_{s_{k+1}} e'$.

    P2 If $e, e' \in A_{k+1}$, then $e \preceq_{s_k} e' \Leftrightarrow e \preceq_{s_{k+1}} e'$.

    P3 If $e \in B_{k+1}$ and $e' \in A_{k+1}$, then $e \preceq_{s_{k+1}} e'$.

    P4 If $e \in B_{k+1}$, then $e \preceq_{s_{k+1}} pos_{s'}(k+1)$.

    P5 If $e \in A_{k+1}$, then $pos_{s'}(k+1) \preceq_{s_{k+1}} e$.

    P6 $pos_{s'}(k+1) \preceq_{s_{k+1}} pos_{s'}(k+1)$

For each s_i, $i \leq n$, we have the respective prefixes of s_i and s' consisting of the first i events are equal. The proof is a straightforward induction on the length of traces and is omitted.

Most of the proof that s_{k+1} is an asynchronous trace is routine and omitted. Assuming s_k is an asynchronous trace, we verify that $(\forall e, e' \in E_{s_{k+1}})(e \curvearrowleft_{s_{k+1}} e' \Rightarrow e \preceq_{s_{k+1}} e')$. We have $e \curvearrowleft_{s_{k+1}} e' \therefore e \curvearrowleft_{s_k} e' \therefore e \preceq_{s_k} e'$. We know e, e' could be members of the following 3 sets B_{k+1}, A_{k+1}, $\{pos_{s'}(k+1)\}$. Note that $B_{k+1} \cup A_{k+1} \cup \{pos_{s'}(k+1)\} = E_{s_k}$. We have the following cases.

- If $e, e' \in B_{k+1}$, then $e \preceq_{s_{k+1}} e'$ by (P1).
- If $e, e' \in A_{k+1}$, then $e \preceq_{s_{k+1}} e'$ by (P2).
- If $e \in B_{k+1}$ and $e' \in A_{k+1}$, then $e \preceq_{s_{k+1}} e'$ by (P3).
- If $e \in B_{k+1}$ and $e' = pos_{s'}(k+1)$, then $e \preceq_{s_{k+1}} e'$ by (P4).

- If $e = pos_{s'}(k+1)$ and $e' \in A_{k+1}$, then $e \preceq_{s_{k+1}} e'$ by (P5).
- If $e \in A_{k+1}$ and $e' \in B_{k+1}$, then $pos_{s_k}(k+1) \preceq_{s_k} e$ and $e' \preceq_{s_k} pos_{s_k}(k)$ so since $\preceq_{s_k}$ is transitive, $e' \preceq_{s_k} e$. However, we also have $e \preceq_{s_k} e'$; since $\preceq_{s_k}$ is antisymmetric, $e = e'$. This contradicts $e \curvearrowleft_{s_k} e'$.
- If $e = pos_{s'}(k+1)$ and $e' \in B_{k+1}$, then $e' \preceq_{s_k} pos_{s_k}(k)$. Since $pos_{s_k}(k) \preceq_{s_k} pos_{s'}(k+1)$, we get $e' \preceq_{s_k} e$. We then follow the same reasoning as the previous case.
- If $e \in A_{k+1}$ and $e' = pos_{s'}(k+1)$, then $pos_{s_k}(k+1) \preceq_{s_k} e$ and $e \neq pos_{s'}(k+1)$. We have $e \curvearrowleft_{s_k} e' \therefore e \curvearrowleft_{s} e' \therefore e \curvearrowleft_{s'} e'$. Since the prefixes of s_k and s' consisting of the first k events are equal, then $e \in B_{k+1}$. This contradicts $e \in A_{k+1}$.

We will require the following facts in the sequel. Note that for any preorder $\preceq$, there is an associated strict partial order $\prec$, defined as $a \prec b$ if and only if $a \preceq b$ and $b \npreceq a$.

Fact 1. If $e \prec_{s_k} e'$ and $k < pos_{s_k}^{-1}(e) < pos_{s_k}^{-1}(e')$, then $e \prec_s e'$.

The proof is by induction on k. The base case $(k = 0)$ is trivial as $s_0 = s$. For the inductive step, let $e \prec_{s_{k+1}} e'$ and $k+1 < pos_{s_{k+1}}^{-1}(e) < pos_{s_{k+1}}^{-1}(e')$. If $s_{k+1} = s_k$, then we apply the inductive hypothesis. Otherwise, we find that $pos_{s_{k+1}}(k+1) \prec_{s_{k+1}} e \prec_{s_{k+1}} e'$. So, $pos_{s'}(k+1) \prec_{s_{k+1}} e \prec_{s_{k+1}} e'$. From the definition of $\preceq_{s_{k+1}}$, we get $e, e' \in A_{k+1}$. Using (P2) yields $e \prec_{s_k} e'$. By definition of A_{k+1}, we also find that $k < pos_{s_k}^{-1}(e) < pos_{s_k}^{-1}(e')$. We then apply the induction hypothesis.

Fact 2. If $pos_{s_{k+1}}(k+1)$ is an output, then all events in $\{pos_{s_k}(i) \mid k+1 \leq i < pos_{s_k}^{-1}(pos_{s_{k+1}}(k+1))\}$ are outputs.

The proof is as follows. Suppose to the contrary that an event e_0 at position i in s_k, such that $k+1 \leq i < pos_{s_k}^{-1}(pos_{s_{k+1}}(k+1))$, is an input. Using Fact 1, $e_0 \preceq_s pos_{s_{k+1}}(k+1)$. We also know that the respective prefixes of s_k and s' consisting of the first k events are equal and $pos_{s_{k+1}}(k+1) = pos_{s'}(k+1)$. Since $e_0 \neq pos_{s_{k+1}}(k+1)$ and $k+1 \leq pos_{s_k}^{-1}(e_0)$, it follows that $pos_{s_{k+1}}(k+1) \preceq_{s'} e_0$. In summary, we have $e_0 \preceq_s pos_{s_{k+1}}(k+1)$ and e_0 is an input and $pos_{s_{k+1}}(k+1)$ is an output and $e_0 \npreceq_{s'} pos_{s_{k+1}}(k+1)$. This contradicts $s' \lesssim^0 s$.

We can now verify that $s_{k+1} \lesssim s_k$. The bijection between E_{s_k} and $E_{s_{k+1}}$ is the identity. The event at position $k+1$ in s_{k+1} occurs at position x in s_k where $k+1 \leq x \leq n$. If $pos_{s_{k+1}}(k+1)$

is an input, then making it earlier is consistent with $\lesssim^{sm}$. If it is an output, then all events in $E_0 = \{pos_{s_k}(i) \mid k+1 \leq i < x\}$ are outputs by Fact 2 and therefore making all events in E_0 later than $pos_{s_{k+1}}(k+1)$ is consistent with $\lesssim$; that is, if $|E_0| = m$, there are m traces s_j, $j \in \{1, \ldots, m\}$ such that $s_{k+1} = s_m \lesssim^{sm} s_{m-1} \lesssim^{sm} \ldots \lesssim^{sm} s_1 \lesssim^{sm} s_k$.

Now, we prove the right to left implication. Let $s' \lesssim s$. Without loss of generality, we assume the bijection in Definition 5.31 is an identity, i.e. $E_s = E_{s'}$, $\lambda_s = \lambda_{s'}$ and $\curvearrowleft_s = \curvearrowleft_{s'}$. From $s' \lesssim s$, we know that there are n traces s_i, $i \in \{1, \ldots, n\}$ such that $s' = s_n \lesssim^{sm} s_{n-1} \lesssim^{sm} \ldots \lesssim^{sm} s_1 \lesssim^{sm} s_0 = s$. Again, without loss of generality, we assume the bijection between each s_i and s_{i+1} is an identity. We need to verify that the conditions of Definition 5.32 hold. It is sufficient to show that: if $e \preceq_{s_i} e'$ and e is an input and e' is an output, then $e \preceq_{s_{i+1}} e'$. This follows from Definition 5.31. ■

In the sequel, we use the preorders $\lesssim$ and $\lesssim^0$ interchangeably.

Another requirement that asynchronous processes must satisfy is *O-completeness*. This condition, used in the game model of asynchronous concurrency [GM08], models the fact that the environment (or the Opponent in the terminology of Game Semantics) cannot be controlled. Therefore, the process should expect any legal input at any time.

Definition 5.34 (O-completeness). *A process $\sigma : A$ consisting of asynchronous traces is* O-complete *if the following condition holds. If $s \in \sigma$ and there is $t \in \Gamma(A)$ such that s is a prefix of t and $\lceil t \rceil = \lceil s \rceil \cdot i$, where i is an input, then $t \in \sigma$.*

It is convenient to define an *O-extension* preorder $\leqslant$ on $\Gamma(A)$ as the least reflexive and transitive relation satisfying for asynchronous traces s and t, $t \leqslant s$ if s is a prefix of t and $\lceil t \rceil = \lceil s \rceil \cdot i$, where i is an input.

We can now introduce asynchronous processes.

Definition 5.35 (Asynchronous process). *An* asynchronous process *over signature A is a nonempty, O-complete, prefix and $\lesssim$-downward closed set of asynchronous traces.*

Processes that are $\lesssim$-closed are said to be *saturated*.

Interaction and composition of asynchronous processes are defined in the same way as for causal processes. The only difference is that we require interaction traces to be asynchronous;

that is, we define the set $int_A(A, B, C)$ of interaction traces over A, B and C as $\Gamma((A \Rightarrow B) \Rightarrow C)$. The definition of tensor product of processes similarly only contains asynchronous interleavings of traces.

Note that although all asynchronous processes are also morphisms in $\mathbf{SynProc}_P$, they do not form a subcategory thereof. The identity for causal processes, in general, is synchronous, instantly replicating any input at one end as an output on the other. Physically, it corresponds to a set of wires directly connecting input and output. Conceptually, it is an instantaneous version of the game semantic copycat strategy. Hence, the synchronous causal identity is not an asynchronous process. However, asynchronous processes have their own notion of identity, similar to the copycat strategy in asynchronous games [GM08].

Define $\Gamma^{\text{alt}}(A)$ as the set of alternating asynchronous traces over A, i.e., those traces where no two consecutive events have the same polarity.

Definition 5.36 (Alternating copycat)**.** *The alternating copycat over A, written cc_A, is the set of traces $u \in \Gamma^{alt}(A \Rightarrow A')$ satisfying the following condition. There is an injection*

$$\text{iccopy}_u : \{e \in E_u \mid \pi \circ \lambda_u(e) = o\} \to \{e \in E_u \mid \pi \circ \lambda_u(e) = i\}$$

assigning to each output an input copy *such that for each output $e \in E_u$,*

C1 $\text{iccopy}_u(e) \preceq_u e$ *and there is no $e' \in E_u$ satisfying $e' \neq e$ and $e' \neq$* $\text{iccopy}_u(e)$ *and* $\text{iccopy}_u(e) \preceq_u e' \preceq_u e$ *and*

C2 $\lambda_u(\text{iccopy}_u(e)) = inl(a)$ *iff* $\lambda_u(e) = inr(a)$ *and* $\lambda_u(\text{iccopy}_u(e)) = inr(a)$ *iff* $\lambda_u(e) = inl(a)$ *and*

C3

- *if $\lambda_u(e) \in I_A$ or $\lambda_u(\text{iccopy}_u(e)) \in I_{A'}$, then $\text{iccopy}_u(e) \curvearrowleft_u e$*
- *if $\lambda_u(e) \notin I_A$ and $\lambda(\text{iccopy}_u(e)) \notin I_{A'}$, then there are $e_1, e_2 \in E_u$ such that $e_1 \curvearrowleft_u e$ and $e_2 \curvearrowleft_u \text{iccopy}_u(e)$ and $e_1 = \text{iccopy}_u(e_2)$.*

Note that for each $u \in \Gamma^{\text{alt}}(A \Rightarrow A')$, if iccopy_u exists, it is necessarily unique by (C1).

The copycat is adapted to the asynchronous framework by allowing the output to be delayed.

Definition 5.37 (Asynchronous identity). *The asynchronous identity on A is the set of all asynchronous traces u on $A \Rightarrow A'$ for which there exists an injection*

$$\text{icopy}_u : \{e \in E_u \mid \pi \circ \lambda_u(e) = o\} \to \{e \in E_u \mid \pi \circ \lambda_u(e) = i\}$$

assigning to each output an input copy *such that for each output $e \in E_u$,*

I1 $\text{icopy}_u(e) \preceq_u e$ *and*

I2 $\lambda_u(\text{icopy}_u(e)) = inl(a)$ *iff* $\lambda_u(e) = inr(a)$ *and* $\lambda_u(\text{icopy}_u(e)) = inr(a)$ *iff* $\lambda_u(e) = inl(a)$ *and*

I3
- *if $\lambda_u(e) \in I_A$ or $\lambda_u(\text{icopy}_u(e)) \in I_{A'}$, then* $\text{icopy}_u(e) \curvearrowleft_u e$
- *if $\lambda_u(e) \notin I_A$ and $\lambda(\text{icopy}_u(e)) \notin I_{A'}$, then there are $e_1, e_2 \in E_u$ such that $e_1 \curvearrowleft_u e$ and $e_2 \curvearrowleft_u \text{icopy}_u(e)$ and $e_1 = \text{icopy}_u(e_2)$.*

Note that the inverse of icopy_u is a partial function, denoted by ocopy_u, that maps inputs to their *output copies*.

Theorem 5.38. *Asynchronous processes form a closed symmetric monoidal category that we call* **AsyProc**.

The proof is detailed in the next section.

5.2.1 A Category of Asynchronous Processes

Intuitively, the category **AsyProc** is equivalent to the full subcategory of $\mathcal{G}$—the category of multi-threaded saturated strategies [GM08, pp. 14–18]—whose objects consist of questions only.

We first show that composition is well-defined.

Lemma 5.39. *Composition preserves saturation.*

Proof. A similar result appears in [GM08]. We adopt the same strategy here. We need to show that if $v' \lesssim v$ and $v \in \sigma;\tau$, then $v' \in \sigma;\tau$. By Definition 5.31, we know that $v' \lesssim v$ implies

there are traces v_i, $i \in \{1, \dots, n-1\}$ such that $v' = v_n \lesssim^{sm} v_{n-1} \lesssim^{sm} \dots \lesssim^{sm} v_1 \lesssim^{sm} v$. So, it is sufficient to show: if $v' \lesssim^{sm} v$ and $v \in \sigma;\tau$, then $v' \in \sigma;\tau$

In the following, i is an input event, o is an output event, e is an arbitrary event, and u_B is a sequence of B-events.

1. Let $v = v_1 \cdot e \cdot i \cdot v_2 \in \sigma;\tau$. It follows that there exists $u = u_1 \cdot e \cdot u_B \cdot i \cdot u_2 \in \sigma \between \tau$ satisfying $u \restriction A + C = v$ and $u_1 \restriction A + C = v_1$ and $u_2 \restriction A + C = v_2$. Making i earlier in u_B still results in interaction traces. This is possible because σ and τ are saturated. So, if i is a C-event we call upon τ and if it is an A-event we use σ. Moreover, if i is a C-event, it cannot be caused by B-events, and if it is an A-event, it still cannot be caused by B-events because it is an input. Hence, $u = u_1 \cdot e \cdot i \cdot u_B \cdot u_2 \in \sigma \between \tau$. At this point, if both e and i are A-events (respectively C-events) and $e \not\curvearrowright i$, then by the saturation of σ (respectively τ), we get $u_1 \cdot i \cdot e \cdot u_B \cdot u_2 \in \sigma \between \tau$. Otherwise, $u_1 \cdot i \cdot e \cdot u_B \cdot u_2 \in \sigma \between \tau$, by the definition of interaction (Definition 3.12). We conclude that, in all cases, $v_1 \cdot i \cdot e \cdot v_2 \in \sigma;\tau$.

2. Let $v = v_1 \cdot o \cdot e \cdot v_2 \in \sigma;\tau$. It follows that there exists $u = u_1 \cdot o \cdot u_B \cdot e \cdot u_2 \in \sigma \between \tau$ satisfying $u \restriction A + C = v$ and $u_1 \restriction A + C = v_1$ and $u_2 \restriction A + C = v_2$. Making o later in u_B still results in interaction traces. This is possible because σ and τ are saturated. So, if o is a C-event we call upon τ and if it is an A-event we use σ. Moreover, if o is a A-event, it cannot cause B-events, and if it is an C-event, it still cannot cause B-events because it is an output. Hence, $u = u_1 \cdot u_B \cdot o \cdot e \cdot u_2 \in \sigma \between \tau$. At this point, if both o and e are A-events (respectively C-events) and $o \not\curvearrowright e$, then by the saturation of σ (respectively τ), we get $u_1 \cdot u_B \cdot e \cdot o \cdot u_2 \in \sigma \between \tau$. Otherwise, $u_1 \cdot u_B \cdot e \cdot o \cdot u_2 \in \sigma \between \tau$, by the definition of interaction (Definition 3.12). We conclude that, in all cases, $v_1 \cdot e \cdot o \cdot v_2 \in \sigma;\tau$. ■

Lemma 5.40. *If $\sigma : A \to B$ and $\tau : B \to C$ are asynchronous processes, then $\sigma;\tau$ is an asynchronous process.*

Proof. We need to check that $\sigma;\tau$ is prefix-closed, saturated and O-complete. The proof that $\sigma;\tau$ is prefix-closed is exactly like the proof of Lemma 5.15. We have proved that $\sigma;\tau$ is saturated in Lemma 5.39. We now prove that $\sigma;\tau$ is O-complete.

Let $v \in \sigma;\tau$. Suppose $u \in \sigma \between \tau$ such that $u \restriction A + C = v$. So, $u \restriction A + B \in \sigma$ and $u \restriction B + C \in \tau$.

Let $w = v \cdot i \in \Gamma(A \Rightarrow C)$. By definition of asynchronous trace, we have the following cases.

1. There is an event $e' \in E_v$ such that $e' \curvearrowleft_w i$ and $\lambda_w(e') \vdash_{A \Rightarrow C} \lambda_w(i)$. If i is a C-event then $(u \upharpoonright B + C) \cdot i \in \Gamma(B \Rightarrow C)$ because $e' \in E_v \Leftrightarrow e' \in E_{u \upharpoonright B+C}$ and $e' \curvearrowleft_w i \Leftrightarrow e' \curvearrowleft_{u \upharpoonright B+C \cdot i} i$ and $\lambda_w(e') \vdash_{A \Rightarrow C} \lambda_w(i) \Leftrightarrow \lambda_{u \upharpoonright B+C \cdot i}(e') \vdash_{B \Rightarrow C} \lambda_{u \upharpoonright B+C \cdot i}(i)$. Since τ is O-complete, we have $(u \upharpoonright B + C) \cdot i \in \tau$. So, $u \cdot i \in \sigma \between \tau$. Finally, we get $v \cdot i \in \sigma ; \tau$. If i is an A-event, there are two cases.

   - If e' is an A-event we use the same reasoning as the first case and the fact that σ is O-complete.
   - If e' is a C-event, then, by definition of interaction, there is a B-event b such that $e' \curvearrowleft_{u \upharpoonright B+C} b$ and $b \curvearrowleft_{u \upharpoonright A+B \cdot i} i$. We then use the same reasoning as the previous case.

2. Or i is initial. If i is an A-event, we use the fact that σ is O-complete. If i is an C-event, we use the fact that τ is O-complete. ■

Most required proofs are exactly the same as those we presented for $\mathbf{SynProc}_P$. In particular, the proof of associativity of composition uses the same strategy as the proof of Lemma 5.20. However, we use the following variant of Lemma 3.26 and Lemma 5.17.

Lemma 5.41. *Let s be an asynchronous interaction trace over $int_A(A, B, C)$ and t be an asynchronous trace over $C \to D$. We have $(s \upharpoonright_B^{A+C}) \between t = (s \between t) \upharpoonright_B^{A+C+D}$.*

Proof. First, we prove $(s \upharpoonright_B^{A+C}) \between t \subseteq (s \between t) \upharpoonright_B^{A+C+D}$. Let $u \in (s \upharpoonright_B^{A+C}) \between t$; that is, $u \in int_A(A, C, D)$ and $u \upharpoonright A + C = s \upharpoonright A + C$ and $u \upharpoonright C + D = t$. To show that $u \in (s \between t) \upharpoonright_B^{A+C+D}$, we construct a trace $v \in int_A(A, B, C, D)$ such that $v \upharpoonright A + C + D = u$ and $v \upharpoonright A + B + C = s$ and $v \upharpoonright C + D = t$ as follows.

Without loss of generality, we assume that $E_{s \upharpoonright A+C} = E_{u \upharpoonright A+C}$. This allows us to simplify the proof by avoiding trace equivalence maps.

We define the trace v exactly as in the proof of Lemma 5.17.

- $E_v = E_{u \upharpoonright A} + E_{s \upharpoonright B} + E_{u \upharpoonright C} + E_{u \upharpoonright D}$. In the sequel, we will omit injections on events; for example, we will write e where it should be $in_j(e), j \in \{1, 2, 3, 4\}$. This is possible because

the sets $E_{u\upharpoonright A}$, $E_{s\upharpoonright B}$, $E_{u\upharpoonright C}$ and $E_{u\upharpoonright D}$ are disjoint. It should improve readability at the cost of abusing notation.

- $\lambda_v = \lambda_{u\upharpoonright A} + \lambda_{s\upharpoonright B} + \lambda_{u\upharpoonright C} + \lambda_{u\upharpoonright D}$
- We define $\preceq_v$ in two steps. Let $\preceq_1$ be the relation satisfying the following.

  P1 $(\forall e, e' \in E_{u\upharpoonright A} + E_{u\upharpoonright C} + E_{u\upharpoonright D})(e \preceq_u e' \Leftrightarrow e \preceq_1 e')$

  P2 $(\forall e, e' \in E_{u\upharpoonright A} + E_{s\upharpoonright B} + E_{u\upharpoonright C})(e \preceq_s e' \Leftrightarrow e \preceq_1 e')$

  Note that the only events that are not comparable using $\preceq_1$ are B-events with respect to D-events. We define a second relation $\preceq_2$ as follow. For all $e \in E_{s\upharpoonright B}$ and $e' \in E_{u\upharpoonright D}$

  P3 $(\exists e'' \in E_v)(e' \preceq_1 e''$ and $e'' \preceq_1 e)$ if and only if $e' \preceq_2 e$

  P4 $(\exists e'' \in E_v)(e \preceq_1 e''$ and $e'' \preceq_1 e')$ or $(\forall e'' \in E_v)\big((e \not\preceq_1 e''$ or $e'' \not\preceq_1 e')$ and $(e' \not\preceq_1 e''$ or $e'' \not\preceq_1 e)\big)$ if and only if $e \preceq_2 e'$

  We then set $\preceq_v$ to be the union of $\preceq_1$ and $\preceq_2$.

- We define $\curvearrowleft_v$ as follows. For all $e, e' \in E_v$,

  J1 if $e, e' \in E_{s\upharpoonright A}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e' \Leftrightarrow e \curvearrowleft_u e'$.

  J2 if $e, e' \in E_{s\upharpoonright B}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e$.

  J3 if $e, e' \in E_{s\upharpoonright C}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e' \Leftrightarrow e \curvearrowleft_u e'$.

  J4 if $e, e' \in E_{s\upharpoonright D}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_u e'$.

  J5 if $e \in E_{s\upharpoonright B}$ and $e' \in E_{s\upharpoonright A}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e'$.

  J6 if $e \in E_{s\upharpoonright C}$ and $e' \in E_{s\upharpoonright B}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_s e'$.

  J7 if $e \in E_{u\upharpoonright D}$ and $e' \in E_{u\upharpoonright C}$, then $e \curvearrowleft_v e' \Leftrightarrow e \curvearrowleft_u e'$.

  J8 The only pointers in $\curvearrowleft_v$ are those specified by (J1)–(J7); that is, if $[e \in E_{u\upharpoonright D}$ and $e' \in E_{s\upharpoonright B}]$ or $[e \in E_{u\upharpoonright D}$ and $e' \in E_{s\upharpoonright A}]$ or $[e \in E_{s\upharpoonright C}$ and $e' \in E_{u\upharpoonright D}]$ or $[e \in E_{s\upharpoonright C}$ and $e' \in E_{s\upharpoonright A}]$ or $[e \in E_{s\upharpoonright B}$ and $e' \in E_{u\upharpoonright D}]$ or $[e \in E_{s\upharpoonright B}$ and $e' \in E_{s\upharpoonright C}]$ or $[e \in E_{s\upharpoonright A}$ and $e' \in E_{u\upharpoonright D}]$ or $[e \in E_{s\upharpoonright A}$ and $e' \in E_{s\upharpoonright C}]$ or $[e \in E_{s\upharpoonright A}$ and $e' \in E_{s\upharpoonright B}]$, then $e \not\curvearrowleft_v e'$.

We have shown in the proof of Lemma 3.26 that $\preceq_v$ is a total preorder and that v has singular events. We also proved that v is a justified in Lemma 5.17.

We only need to verify that $\preceq_v$ is antisymmetric; i.e., for all $e, e' \in E_v$, if $e \preceq_v e'$ and $e' \preceq_v e$, then $e = e'$. The proof is by case analysis. There are sixteen cases corresponding to the label assignments of e and e'. Cases (A, A), (A, B), (A, C), (A, D), (B, A), (B, B), (B, C), (C, A), (C, B), (C, C), (C, D), (D, A), (D, C), (D, D) are similar, so we only prove it for the case where $\lambda_v(e), \lambda_v(e') \in L_A$. We have $e \preceq_v e' \Leftrightarrow e \preceq_1 e' \Leftrightarrow e \preceq_s e'$ by (P1). Similarly, $e' \preceq_v e \Leftrightarrow e' \preceq_1 e \Leftrightarrow e' \preceq_s e$ by (P1). We get $e \preceq_s e'$ and $e' \preceq_s e$. So, $e = e'$ because $\preceq_s$ is antisymmetric.

The remaining two cases are symmetric so we only consider one of them. Let $\lambda_v(e) \in L_B$ and $\lambda_v(e') \in L_D$. We have

- $e \preceq_v e' \Leftrightarrow e \preceq_2 e'$ iff
  - $(\exists e_0 \in E_v)(e \preceq_1 e_0$ and $e_0 \preceq_1 e')$ (let us refer to this by $(\star)$) or
  - $(\forall e'' \in E_v)\big((e \not\preceq_1 e''$ or $e'' \not\preceq_1 e')$ and $(e' \not\preceq_1 e''$ or $e'' \not\preceq_1 e)\big)$ (let us refer to this by $(\blacktriangle)$)
- $e' \preceq_v e \Leftrightarrow e' \preceq_2 e \Leftrightarrow (\exists e_1 \in E_v)(e' \preceq_1 e_1$ and $e_1 \preceq_1 e)$ (let us refer to this by $(\blacklozenge)$)

We have $(\blacktriangle)$ and $(\blacklozenge)$ is a contradiction. Suppose $(\star)$ and $(\blacklozenge)$. We have $(\star) \Leftrightarrow (\exists e_0 \in E_v)(e \preceq_s e_0$ and $e_0 \preceq_u e')$ and $(\blacklozenge) \Leftrightarrow (\exists e_1 \in E_v)(e' \preceq_u e_1$ and $e_1 \preceq_s e)$. We also have $e_0, e_1 \in E_{u\restriction A+C} = E_{s\restriction A+C}$. Since $\preceq_s$ and $\preceq_u$ are total orders, we have either $e_0 \preceq_s e_1$ and $e_0 \preceq_u e_1$ or $e_1 \preceq_s e_0$ and $e_1 \preceq_u e_0$. It follows that $e \preceq_s e_0 \preceq_s e_1$ and $e_0 \preceq_s e_1 \preceq_s e$ or $e_1 \preceq_u e_0 \preceq_u e'$ and $e' \preceq_u e_1 \preceq_u e_0$, both of which are contradictions.

The rest of the proof is similar to Lemma 5.17. ■

We will now present the lemmas related to the identity and saturation.

Lemma 5.42. *The alternating copycat is contained within the identity, i.e.* $cc_A \subseteq id_A$.

The proof is immediate from Definition 5.36.

Lemma 5.43. *The asynchronous identity is an asynchronous process.*

Proof. The fact that the identity is prefix-closed and O-complete follows directly from its definition. We prove that the identity is saturated.

Suppose $s \in id_A$ and $s' \lesssim s$; that is, there is a bijection $\phi : E_s \to E_{s'}$ such that the following conditions hold.

S1 $\lambda_s = \lambda_{s'} \circ \phi$

S2 $(\forall e, e' \in E_s)(e \curvearrowleft_s e'$ if and only if $\phi(e) \curvearrowleft_{s'} \phi(e'))$

S3 If $e \preceq_s e'$ and e is an input and e' is an output, then $\phi(e) \preceq_{s'} \phi(e')$

We set $\text{icopy}_{s'}(e) = \phi(\text{icopy}_s(\phi^{-1}(e)))$. We need to prove that s' with $\text{icopy}_{s'}$ satisfies conditions (I1)–(I3) of Definition 5.37. Take an output e in s' and let $e' = \text{icopy}_s(\phi^{-1}(e))$. So, $\text{icopy}_{s'}(e) = \phi(e')$.

- We first show that $\text{icopy}_{s'}(e) \preceq_{s'} e$. We have by Definition 5.37, $\text{icopy}_s(\phi^{-1}(e)) \preceq_s \phi^{-1}(e)$. So, $e' \preceq_s \phi^{-1}(e)$. Since e' is an input and $\phi^{-1}(e)$ is an output, we use (S3) to find $\phi(e') \preceq_s \phi(\phi^{-1}(e))$. Hence, $\text{icopy}_{s'}(e) \preceq_{s'} e$.

- We have $\lambda_s(e') = inl(a)$ iff $\lambda_s(\phi^{-1}(e)) = inr(a)$, $a \in L_A$. By (S1), $\lambda_s(e') = inl(a)$ iff $\lambda_{s'}(\phi(e')) = inl(a)$ and $\lambda_s(\phi^{-1}(e)) = inr(a)$ iff $\lambda_{s'}(e) = inr(a)$. So, $\lambda_{s'}(\phi(e')) = inl(a)$ iff $\lambda_{s'}(e) = inr(a)$. Using the same reasoning, we can prove $\lambda_{s'}(\phi(e')) = inr(a)$ iff $\lambda_{s'}(e) = inl(a)$.

- Let $\lambda_{s'}(\phi(e')) \in I_{A'}$ or $\lambda_{s'}(e) \in I_A$. By (S1), we get $\lambda_s(e') \in I_{A'}$ or $\lambda_s(\phi^{-1}(e)) \in I_A$. Since $s \in id_A$, we use (I3) in Definition 5.37 to obtain $e' \curvearrowleft_s \phi^{-1}(e)$. We then use (S2) to find $\phi(e') \curvearrowleft_{s'} e$.

- Let $\lambda_{s'}(\phi(e')) \notin I_{A'}$ and $\lambda_{s'}(e) \notin I_A$. By (S1), we get $\lambda_s(e') \notin I_{A'}$ and $\lambda_s(\phi^{-1}(e)) \notin I_A$. Since $s \in id_A$, we use (I3) in Definition 5.37 to find that there are $e_1, e_2 \in E_s$ such that $e_1 \curvearrowleft_s e'$ and $e_2 \curvearrowleft_s \phi^{-1}(e)$ and $e_2 = \text{icopy}_s(e_1)$. We then use (S2) to find $\phi(e_1) \curvearrowleft_{s'} \phi(e')$ and $\phi(e_2) \curvearrowleft_{s'} e$ and $\text{icopy}_{s'}(\phi(e_1)) = \phi(\text{icopy}_s(\phi^{-1}(\phi(e_1)))) = \phi(\text{icopy}_s(e_1)) = \phi(e_2)$. ■

Lemma 5.44. *Let $\sigma : A \to B$ be an asynchronous process and $cc_B : B \to B'$ be the alternating copycat over B. If $u \in \sigma \lightning cc_B$ and $u \upharpoonright B + B'$ has even length, then $u \upharpoonright A + B' \lesssim^0 u \upharpoonright A + B$.*

Proof. Let $u \in \sigma \lightning cc_B$ and $u \upharpoonright B + B'$ has even length. Following Definition 5.32, we first need to provide a bijection $\phi : E_{u\upharpoonright A+B} \to E_{u\upharpoonright A+B'}$ such that $\lambda_{u\upharpoonright A+B} = \lambda_{u\upharpoonright A+B'} \circ \phi$ and $(\forall e, e' \in E_{u\upharpoonright A+B})(e \curvearrowleft_{u\upharpoonright A+B} e'$ if and only if $\phi(e) \curvearrowleft_{u\upharpoonright A+B'} \phi(e'))$.

Define ccopy $=$ $\text{iccopy}_{u\restriction B+B'} \cup \text{iccopy}^{-1}_{u\restriction B+B'}$. Note that the domains of $\text{iccopy}_{u\restriction B+B'}$ and $\text{iccopy}^{-1}_{u\restriction B+B'}$ are disjoint and so are their codomains. Moreover, the domain and codomain of ccopy is equal to $E_{u\restriction B+B'}$. By Definition 5.36 (copycat), we know that the input copy of each output in $u \restriction B + B'$ is the event that directly precedes it. Since $u \restriction B + B'$ has even length, its last event is an output, and so each input has an output copy. Therefore, $\text{iccopy}^{-1}_{u\restriction B+B'}$ is an injection. So, ccopy is a bijection. Additionally, note that ccopy is an involution.

Define $\phi : E_{u\restriction A+B} \to E_{u\restriction A+B'}$ as follows, for all $e \in E_{u\restriction A+B}$.

$$\phi(e) = \begin{cases} e & \text{if } \lambda_u(e) \in L_A \\ \text{ccopy}(e) & \text{if } \lambda_u(e) \in L_B. \end{cases}$$

Its inverse ϕ^{-1} is defined as follows, for all $e \in E_{u\restriction A+B'}$.

$$\phi^{-1}(e) = \begin{cases} e & \text{if } \lambda_u(e) \in L_A \\ \text{ccopy}(e) & \text{if } \lambda_u(e) \in L_{B'}. \end{cases}$$

It is clear that ϕ is a bijection.

We first prove that $\lambda_{u\restriction A+B} = \lambda_{u\restriction A+B'} \circ \phi$. Let $e \in E_{u\restriction A+B}$. If $\lambda_{u\restriction A+B}(e) \in L_A$, then $\phi(e) = e$. We have $\lambda_{u\restriction A+B}(e) = a \in L_A \therefore \lambda_u(e) = a \therefore \lambda_{u\restriction A+B'}(e) = a \therefore \lambda_{u\restriction A+B'}(\phi(e)) = a$. If $\lambda_{u\restriction A+B}(e) \in L_B$, then $\phi(e) = \text{ccopy}(e)$. We have $\lambda_{u\restriction A+B}(e) = b \in L_B \therefore \lambda_u(e) = inl(inr((b)) \therefore \lambda_{u\restriction B+B'}(e) = inl(b)$. Using Definition 5.36 (copycat), we get $\lambda_{u\restriction B+B'}(\text{ccopy}(e)) = inr(b) \therefore \lambda_u(\phi(e)) = inr(b) \therefore \lambda_{u\restriction A+B'}(\phi(e)) = b$.

Next we show that $(\forall e, e' \in E_{u\restriction A+B})(e \curvearrowleft_{u\restriction A+B} e' \Leftrightarrow \phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e'))$. For the left to right implication, suppose $e, e' \in E_{u\restriction A+B}$ and $e \curvearrowleft_{u\restriction A+B} e'$. We have the following cases.

- If $\lambda_u(e), \lambda_u(e') \in L_A$, then $\phi(e) = e$ and $\phi(e') = e'$. We have $e \curvearrowleft_{u\restriction A+B} e' \therefore e \curvearrowleft_u e' \therefore e \curvearrowleft_{u\restriction A+B'} e' \therefore \phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e')$.
- If $\lambda_u(e), \lambda_u(e') \in L_B$, then $\phi(e) = \text{ccopy}(e)$ and $\phi(e') = \text{ccopy}(e')$. We have $e \curvearrowleft_{u\restriction A+B} e' \therefore e \curvearrowleft_u e' \therefore e \curvearrowleft_{u\restriction B+B'} e'$. We then have the following two cases.
  - Let e be an input and e' be an output in $u \restriction B + B'$. Using Definition 5.36, $e \curvearrowleft_{u\restriction B+B'}$

$e' \therefore \mathrm{iccopy}^{-1}(e) \curvearrowleft_{u\restriction B+B'} \mathrm{iccopy}(e')$. So, $\mathrm{ccopy}(e) \curvearrowleft_{u\restriction B+B'} \mathrm{ccopy}(e') \therefore \phi(e) \curvearrowleft_{u} \phi(e') \therefore \phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e')$.

– Let e be an output and e' be an input in $u \restriction B+B'$. Using Definition 5.36, $e \curvearrowleft_{u\restriction B+B'} e' \therefore \mathrm{iccopy}(e) \curvearrowleft_{u\restriction B+B'} \mathrm{iccopy}^{-1}(e')$. So, $\mathrm{ccopy}(e) \curvearrowleft_{u\restriction B+B'} \mathrm{ccopy}(e') \therefore \phi(e) \curvearrowleft_{u} \phi(e') \therefore \phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e')$.

- If $\lambda_u(e) \in L_B$ and $\lambda_u(e') \in L_A$, then $\phi(e) = \mathrm{ccopy}(e)$ and $\phi(e') = e'$. From $e \curvearrowleft_{u\restriction A+B} e' \therefore e \curvearrowleft_{u} e'$, we know e is initial in B and e' is initial in A. By Definition 5.36, $\mathrm{iccopy}(e) \curvearrowleft_{u\restriction B+B'} e$. So, $\mathrm{ccopy}(e) \curvearrowleft_{u\restriction B+B'} e \therefore \phi(e) \curvearrowleft_{u} e$. By definition of composition, we get $\phi(e) \curvearrowleft_{u\restriction A+B'} e'$ and hence $\phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e')$.

For the right to left implication, suppose $e, e' \in E_{u\restriction A+B}$ and $\phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e')$.

- If $\lambda_u(e), \lambda_u(e') \in L_A$, then $\phi(e) = e$ and $\phi(e') = e'$. We have $\phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e') \therefore \phi(e) \curvearrowleft_{u} \phi(e') \therefore \phi(e) \curvearrowleft_{u\restriction A+B} \phi(e') \therefore e \curvearrowleft_{u\restriction A+B} e'$.
- If $\lambda_u(e), \lambda_u(e') \in L_B$, then $\phi(e) = \mathrm{ccopy}(e)$ and $\phi(e') = \mathrm{ccopy}(e')$. We have $\phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e') \therefore \phi(e) \curvearrowleft_{u} \phi(e') \therefore \phi(e) \curvearrowleft_{u\restriction B+B'} \phi(e')$. We then have the following two cases.
  – Let $\phi(e)$ be an input and $\phi(e')$ be an output in $u \restriction B+B'$. Using Definition 5.36, $\phi(e) \curvearrowleft_{u\restriction B+B'} \phi(e') \therefore \mathrm{iccopy}^{-1}(\phi(e)) \curvearrowleft_{u\restriction B+B'} \mathrm{iccopy}(\phi(e'))$. So, $\mathrm{ccopy}(\phi(e)) \curvearrowleft_{u\restriction B+B'} \mathrm{ccopy}(\phi(e')) \therefore \phi^{-1}(\phi(e)) \curvearrowleft_{u} \phi^{-1}(\phi(e')) \therefore e \curvearrowleft_{u\restriction A+B'} e'$.
  – Let $\phi(e)$ be an output and $\phi(e')$ be an input in $u \restriction B+B'$. Using Definition 5.36, $\phi(e) \curvearrowleft_{u\restriction B+B'} \phi(e') \therefore \mathrm{iccopy}(\phi(e)) \curvearrowleft_{u\restriction B+B'} \mathrm{iccopy}^{-1}(\phi(e'))$. So, $\mathrm{ccopy}(\phi(e)) \curvearrowleft_{u\restriction B+B'} \mathrm{ccopy}(\phi(e')) \therefore \phi^{-1}(\phi(e)) \curvearrowleft_{u} \phi^{-1}(\phi(e')) \therefore e \curvearrowleft_{u\restriction A+B'} e'$.
- If $\lambda_u(e) \in L_B$ and $\lambda_u(e') \in L_A$, then $\phi(e) = \mathrm{ccopy}(e)$ and $\phi(e') = e'$. From $\phi(e) \curvearrowleft_{u\restriction A+B'} \phi(e')$, we know $\phi(e)$ is initial in B' and e' is initial in A. By the definition of composition, there exists e_0 such that $\phi(e) \curvearrowleft_{u\restriction B+B'} e_0$ and $e_0 \curvearrowleft_{u\restriction A+B} e'$. By Definition 5.36, $\phi(e) \curvearrowleft_{u\restriction B+B'} \mathrm{iccopy}^{-1}(\phi(e)) \therefore \phi(e) \curvearrowleft_{u\restriction B+B'} \mathrm{ccopy}(\phi(e)) \therefore \phi(e) \curvearrowleft_{u\restriction B+B'} \phi^{-1}(\phi(e)) \therefore \phi(e) \curvearrowleft_{u\restriction B+B'} e$. So, $e = e_0$ and therefore $e \curvearrowleft_{u\restriction A+B} e'$.

Now, we show that the third condition in Definition 5.32 holds, i.e. for all $e, e' \in E_{u\restriction A+B}$ if $e \preceq_{u\restriction A+B} e'$ and e is an input and e' is an output, then $\phi(e) \preceq_{u\restriction A+B'} \phi(e')$. Let $e \preceq_{u\restriction A+B} e'$ and e be an input and e' be an output. We have the following cases.

- If $e, e' \in E_{u\upharpoonright A}$, then $\phi(e) = e$ and $\phi(e') = e'$. We have $e \preceq_{u\upharpoonright A+B} e' \therefore e \preceq_u e' \therefore e \preceq_{u\upharpoonright A+B'} e' \therefore \phi(e) \preceq_{u\upharpoonright A+B'} \phi(e')$.

- If $e, e' \in E_{u\upharpoonright B}$, then $\phi(e) = \text{ccopy}(e)$ and $\phi(e') = \text{ccopy}(e')$. We have $e \preceq_{u\upharpoonright A+B} e' \therefore e \preceq_u e' \therefore e \preceq_{u\upharpoonright B+B'} e' \therefore \text{ccopy}(e) \preceq_{u\upharpoonright B+B'} \text{ccopy}(e')$ [since $\text{ccopy}(e) \preceq_{u\upharpoonright B+B'} e$ and $e' \preceq_{u\upharpoonright B+B'} \text{ccopy}(e')$] $\therefore \text{ccopy}(e) \preceq_u \text{ccopy}(e') \therefore \phi(e) \preceq_u \phi(e')$.

- If $e \in E_{u\upharpoonright A}$ and $e' \in E_{u\upharpoonright B}$, then $\phi(e) = e$ and $\phi(e') = \text{ccopy}(e')$. We have $e \preceq_{u\upharpoonright A+B} e' \therefore e \preceq_u e' \therefore e \preceq_u \text{ccopy}(e')$ [since $e' \preceq_{u\upharpoonright B+B'} \text{ccopy}(e')$] $\therefore e \preceq_{u\upharpoonright A+B'} \text{ccopy}(e') \therefore \phi(e) \preceq_{u\upharpoonright A+B'} \phi(e')$.

- If $e \in E_{u\upharpoonright B}$ and $e' \in E_{u\upharpoonright A}$, then $\phi(e) = \text{ccopy}(e)$ and $\phi(e') = e'$. We have $e \preceq_{u\upharpoonright A+B} e' \therefore e \preceq_u e' \therefore \text{ccopy}(e) \preceq_u e'$ [since $\text{ccopy}(e) \preceq_{u\upharpoonright B+B'} e$] $\therefore \text{ccopy}(e) \preceq_{u\upharpoonright A+B'} e' \therefore \phi(e) \preceq_{u\upharpoonright A+B'} \phi(e')$. ■

Corollary 5.45. *Let $\sigma : A \to B$ be an asynchronous process and $cc_B : B \to B'$ be the alternating copycat over B. If $u \in \sigma \between cc_B$ and $u \upharpoonright B + B'$ has even length, then $u \upharpoonright A + B' \lesssim u \upharpoonright A + B$.*

Proof. Let $u \in \sigma \between cc_B$ and $u \upharpoonright B + B'$ has even length. Using Lemma 5.44, we find $u \upharpoonright A + B' \lesssim^0 u \upharpoonright A + B$. We then use Lemma 5.33 to conclude that $u \upharpoonright A + B' \lesssim u \upharpoonright A + B$. ■

We need the following ancillary notions in the sequel.

Definition 5.46. *Define the preorder $\lesssim^{int}$ on $int_A(A, B, C)$ as the relation satisfying $u' \lesssim^{int} u$ when $u \upharpoonright B = u' \upharpoonright B$ and there is a bijection $\phi : E_u \to E_{u'}$ such that the following conditions hold.*

1. *$\lambda_u = \lambda_{u'} \circ \phi$*
2. *$(\forall e, e' \in E_u)(e \frown_u e'$ if and only if $\phi(e) \frown_u \phi(e'))$*
3. *If $e \preceq_u e'$ and [e is a C-input or a B-input] and [e' is a C-output or a B-output], then $\phi(e) \preceq_{u'} \phi(e')$*
4. *If $e \preceq_u e'$ and [e is a B-output or an A-output] and [e' is a B-input or an A-input], then $\phi(e) \preceq_{u'} \phi(e')$*

Definition 5.47. *Define the preorder $\leqslant^{int}$ on $int_A(A, B, C)$ as the least reflexive and transitive relation satisfying for asynchronous traces u and u', $u \leqslant u'$ when u is a prefix of u' and $\lceil u' \rceil = \lceil u \rceil \cdot e$, where e is a C-input or an A-output.*

In the previous two definitions, we want the preorders to be consistent with the saturation and O-extension preorders on $\Gamma(A \Rightarrow B)$. This accounts for the change in the polarity of A-events.

We can now state the following lemmas.

Lemma 5.48. *Let u and u' be interaction traces in $int_A(A, B, C)$ and $u_0, u_0' \in \Gamma(A \Rightarrow C)$ such that $u \upharpoonright A + C = u_0$ and $u' \upharpoonright A + C = u_0'$. If $u \lesssim^{int} u'$, then $u_0 \lesssim u_0'$.*

Lemma 5.49. *Let u and u' be interaction traces in $int_A(A, B, C)$ and $u_0, u_0' \in \Gamma(A \Rightarrow C)$ such that $u \upharpoonright A + C = u_0$ and $u' \upharpoonright A + C = u_0'$. If $u \leqslant^{int} u'$, then $u_0 \leqslant u_0'$.*

Lemma 5.50. *Let u and u' be interaction traces in $int_A(A, B, C)$. If $u \preccurlyeq u'$, then $u \upharpoonright A + C \preccurlyeq u' \upharpoonright A + C$.*

Lemma 5.51. *The asynchronous identity is an identity to asynchronous processes.*

Proof. Let $\sigma : A \to B$ be an asynchronous process and $id_B : B \to B'$.

We first prove $\sigma \subseteq \sigma; id_B$. Let $s \in \sigma$. We will find an interaction trace $u \in int_A(A, B, B')$ such that $u \upharpoonright A + B = s$ and $u \upharpoonright B + B' \in id_B$ and $u \upharpoonright A + B' = s$. Let u be defined as follows.

- $E_u = E_{s\upharpoonright A} + E_{s\upharpoonright B} + E_{s\upharpoonright B}$. Let us refer to these subsets using injections $in_1 : E_{s\upharpoonright A} \to E_u$, $in_2 : E_{s\upharpoonright B} \to E_u$ and $in_3 : E_{s\upharpoonright B} \to E_u$.
- $\lambda_u = \lambda_{s\upharpoonright A} + \lambda_{s\upharpoonright B} + \lambda_{s\upharpoonright B}$
- The preorder $\preceq_u$ is defined as follows. For all $e, e' \in E_s$,

  P1 if $e, e' \in E_{s\upharpoonright A}$, then $e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_1(e')$

  P2 if $e \in E_{s\upharpoonright A}$ and $e' \in E_{s\upharpoonright B}$, then $[e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_1(e) \preceq_u in_3(e')]$

  P3 if $e \in E_{s\upharpoonright B}$ and $e' \in E_{s\upharpoonright A}$, then $[e \preceq_s e' \Leftrightarrow in_2(e) \preceq_u in_1(e') \Leftrightarrow in_3(e) \preceq_u in_1(e')]$

  P4 if $e, e' \in E_{s\upharpoonright B}$ and $e \neq e'$, then $[e \preceq_s e' \Leftrightarrow in_2(e) \preceq_u in_2(e') \Leftrightarrow in_3(e) \preceq_u in_3(e') \Leftrightarrow in_2(e) \preceq_u in_3(e') \Leftrightarrow in_3(e) \preceq_u in_2(e')]$

P5 $e \in E_{s\restriction B} \Leftrightarrow in_2(e) \preceq_u in_2(e) \Leftrightarrow in_3(e) \preceq_u in_3(e)$

P6 $e \in E_{s\restriction B}$ and $\pi \circ \lambda_s(e) = i \Leftrightarrow in_3(e) \preceq_u in_2(e)$

P7 $e \in E_{s\restriction B}$ and $\pi \circ \lambda_s(e) = o \Leftrightarrow in_2(e) \preceq_u in_3(e)$

- The function $\curvearrowleft_u$ is defined as follows. For all $e, e' \in E_s$,

  J1 if $e, e' \in E_{s\restriction A}$, then $e \curvearrowleft_s e' \Leftrightarrow in_1(e) \curvearrowleft_u in_1(e')$

  J2 if $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $e \curvearrowleft_s e' \Leftrightarrow in_2(e) \curvearrowleft_u in_1(e')$

  J3 if $e, e' \in E_{s\restriction B}$, then $[e \curvearrowleft_s e' \Leftrightarrow in_2(e) \curvearrowleft_u in_2(e') \Leftrightarrow in_3(e) \curvearrowleft_u in_3(e')]$

  J4 $\lambda_s(e) \in I_B \Leftrightarrow in_3(e) \curvearrowleft_u in_2(e)$

  J5 The only pointers in $\curvearrowleft_u$ are those specified by (J1)–(J4); that is,

  – if $e \in E_{s\restriction A}$ and $e' \in E_{s\restriction B}$, then $[in_1(e) \not\curvearrowleft_u in_2(e')$ and $in_1(e) \not\curvearrowleft_u in_3(e')]$,

  – if $e \in E_{s\restriction B}$, then $in_2(e) \not\curvearrowleft_u in_3(e')$,

  – if $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $in_3(e) \not\curvearrowleft_u in_1(e')$.

We first argue that $\preceq_u$ is a total order. As $\preceq_s$ is reflexive, (P1) and (P5) ensure that $\preceq_u$ is reflexive too. Because $\preceq_s$ is total, $\preceq_u$ is total as every two events are comparable: A-events are comparable with A-events by (P1); A-events are comparable with B-events by (P2) and (P3); A-events are comparable with B'-events by (P2) and (P3); B-events are comparable with B-events by (P4) and (P5); B-events are comparable with B'-events by (P4), (P6) and (P7); and B'-events are comparable with B'-events by (P4) and (P5). Next, we show that $\preceq_u$ is transitive. The proof can be achieved via a long case analysis (27 cases). We will consider all cases at once by using subscript variables. Let $in_j(e_1) \preceq_u in_k(e_2)$ and $in_k(e_2) \preceq_u in_l(e_3)$ where $j, k, l \in \{1, 2, 3\}$. By definition of $\preceq_u$, we have $e_1 \preceq_s e_2$ and $e_2 \preceq_s e_3$. Because $\preceq_s$ is transitive, we get $e_1 \preceq_s e_2$. By inspecting the definition of $\preceq_u$, we get $in_j(e_1) \preceq_u in_l(e_3)$.

Next, we need to verify that $\preceq_u$ is antisymmetric, i.e. for all $e, e' \in E_u$, if $e \preceq_u e'$ and $e' \preceq_u e \Rightarrow e = e'$. The proof is by case analysis. There are nine cases corresponding to the label assignments of e, e'.

1. $\lambda(e) \in L_A$ and $\lambda(e') \in L_A$. By (P1), $e \preceq_u e' \Leftrightarrow in_1^{-1}(e) \preceq_s in_1^{-1}(e')$ and $e' \preceq_u e \Leftrightarrow in_1^{-1}(e') \preceq_s in_1^{-1}(e)$. Since $\preceq_s$ is antisymmetric, we conclude that $in_1^{-1}(e) = in_1^{-1}(e')$ and so $e = e'$.

2. $\lambda(e) \in L_A$ and $\lambda(e') \in L_B$. By (P2), $e \preceq_u e' \Leftrightarrow in_1^{-1}(e) \preceq_s in_2^{-1}(e')$ and by (P3) $e' \preceq_u e \Leftrightarrow in_2^{-1}(e') \preceq_s in_1^{-1}(e)$. Since $\preceq_s$ is antisymmetric, we conclude that $in_1^{-1}(e) = in_2^{-1}(e')$. This is a contradiction since in_1 and in_2 have disjoint domains.

3. $\lambda(e) \in L_A$ and $\lambda(e') \in L_{B'}$. We follow the same reasoning as the second case.

4. $\lambda(e) \in L_B$ and $\lambda(e') \in L_A$. We follow the same reasoning as the second case.

5. $\lambda(e) \in L_{B'}$ and $\lambda(e') \in L_A$. We follow the same reasoning as the second case.

6. $\lambda(e) \in L_B$ and $\lambda(e') \in L_{B'}$. There are two cases.

   - $in_2^{-1}(e) \neq in_3^{-1}(e')$. By (P4), $e \preceq_u e' \Leftrightarrow in_2^{-1}(e) \preceq_s in_3^{-1}(e')$ and $e' \preceq_u e \Leftrightarrow in_3^{-1}(e') \preceq_s in_2^{-1}(e)$. Since $\preceq_s$ is antisymmetric, we conclude that $in_2^{-1}(e) = in_3^{-1}(e')$. This is a contradiction since $in_2^{-1}(e) \neq in_3^{-1}(e')$.

   - $in_2^{-1}(e) = in_3^{-1}(e')$. By (P7), $e \preceq_u e' \Rightarrow \pi(\lambda(in_2^{-1}(e))) = o$ and by (P6) $e' \preceq_u e \Rightarrow \pi(\lambda(in_2^{-1}(e))) = i$ thus leading to contradiction.

7. $\lambda(e) \in L_{B'}$ and $\lambda(e') \in L_B$. We follows the same reasoning as the sixth case.

8. $\lambda(e) \in L_B$ and $\lambda(e') \in L_B$. There are two cases.

   - $in_2^{-1}(e) \neq in_2^{-1}(e')$. By (P4), $e \preceq_u e' \Leftrightarrow in_2^{-1}(e) \preceq_s in_2^{-1}(e')$ and $e' \preceq_u e \Leftrightarrow in_2^{-1}(e') \preceq_s in_2^{-1}(e)$. Since $\preceq_s$ is antisymmetric, we conclude that $in_2^{-1}(e) = in_2^{-1}(e')$. This is a contradiction since $in_2^{-1}(e) \neq in_2^{-1}(e')$.

   - $in_2^{-1}(e) = in_2^{-1}(e')$. So $e = e'$.

9. $\lambda(e) \in L_{B'}$ and $\lambda(e') \in L_{B'}$. We use the same strategy as the eighth case.

Next, we need to verify that u is justified, i.e. for all $e \in E_u \setminus \{e \in E_u \mid \lambda_u(e) \in I_{B'}\}$, there is $e' \in E_u$ such that $e' \curvearrowleft_u e$.

- If e is an A-event, we use (J1) and (J2) and the fact that s is justified.

- If e is a noninitial B-event, we use (J3) and the fact that s is justified.

- If e is an initial B-event, we use (J4).

- If e is a noninitial B'-event, we use (J3) and the fact that s is justified.

Additionally, (J5) guarantees that $\frown_u$ is a function, i.e. no event is assigned more than one cause.

Now, we show $(\forall e, e' \in E_u)(e \frown_u e' \Rightarrow (e \preceq_u e \text{ and } \lambda_u(e) \vdash \lambda_u(e'))$. Suppose $e \frown_u e'$. By definition of $\frown_u$, we have the following cases.

- If $e, e' \in E_{u \restriction A}$, then $e \frown_u e' \therefore in_1^{-1}(e) \frown_s in_1^{-1}(e')$ by (J1) $\therefore in_1^{-1}(e) \preceq_s in_1^{-1}(e') \therefore e \preceq_u e'$ by (P1)
- If $e, e' \in E_{u \restriction B}$, then $e \frown_u e' \therefore in_2^{-1}(e) \frown_s in_2^{-1}(e')$ by (J3) $\therefore in_2^{-1}(e) \preceq_s in_2^{-1}(e') \therefore e \preceq_u e'$ by (P4)
- If $e, e' \in E_{u \restriction B'}$, then $e \frown_u e' \therefore in_3^{-1}(e) \frown_s in_3^{-1}(e')$ by (J3) $\therefore in_3^{-1}(e) \preceq_s in_3^{-1}(e') \therefore e \preceq_u e'$ by (P4)
- If $e \in E_{u \restriction B'}$ and $e' \in E_{u \restriction B}$, then $e \frown_u e' \therefore in_3^{-1}(e) = in_2^{-1}(e)$ and $\lambda_s(in_3^{-1}(e)) \in I_B$ by (J4) $\therefore \pi \circ \lambda_s(in_3^{-1}(e)) = i \therefore e \preceq_u e'$ by (P6)
- If $e \in E_{u \restriction B}$ and $e' \in E_{u \restriction A}$, then $e \frown_u e' \therefore in_2^{-1}(e) \frown_s in_1^{-1}(e)$ by (J2) $\therefore in_2^{-1}(e) \preceq_s in_1^{-1}(e) \therefore e \preceq_u e'$ by (P3)

Note that in each case $\lambda_u(e) \vdash \lambda_u(e')$.

Finally, we show that u satisfies the conditions set out above, i.e. $u \restriction B + B' \in id_B$ and $u \restriction A + B = s$ and $u \restriction A + B' = s$. For $u \restriction B + B' \in id_B$, we show that $u \restriction B + B' \in cc_B$. Let

$$\text{iccopy}_{u \restriction B+B'}(e) = \begin{cases} in_3(in_2^{-1}(e)) & \text{if } \lambda_u(e) \in lin_2(B) \\ in_2(in_3^{-1}(e)) & \text{if } \lambda_u(e) \in lin_3(B) \end{cases}$$

We verify that $u \restriction B + B'$ with $\text{iccopy}_{u \restriction B+B'}$ satisfies the conditions of Definition 5.36. Let e be an output in $u \restriction B + B'$.

- We first show that $\text{iccopy}_{u \restriction B+B'}(e) \preceq_{u \restriction B+B'} e$. If e is a B'-event, then $in_3^{-1}(e) \in E_s$ and $\text{iccopy}_{u \restriction B+B'}(e) = in_2(in_3^{-1}(e))$. Note that if e is a B'-output in $u \restriction B + B'$ then $in_3^{-1}(e)$ is an output in s. By (P7), we have $in_2(in_3^{-1}(e)) \preceq_u in_3(in_3^{-1}(e))$. So, $\text{iccopy}_{u \restriction B+B'}(e) \preceq_u e$ and hence, $\text{iccopy}_{u \restriction B+B'}(e) \preceq_{u \restriction B+B'} e$. If e is a B-event, then $in_2^{-1}(e) \in E_s$ and $\text{iccopy}_{u \restriction B+B'}(e) =$

$in_3(in_2^{-1}(e))$. Note that if e is a B-output in $u \restriction B+B'$ then $in_2^{-1}(e)$ is an input in s. By (P6), we have $in_3(in_2^{-1}(e)) \preceq_u in_2(in_2^{-1}(e))$. So, $\text{iccopy}_{u\restriction B+B'}(e) \preceq_u e$ and hence, $\text{iccopy}_{u\restriction B+B'}(e) \preceq_{u\restriction B+B'} e$.

- We now show that there is no $e' \in E_{u\restriction B+B'}$ such that $e' \neq e$ and $e' \neq \text{iccopy}_{u\restriction B+B'}(e)$ and $\text{iccopy}(e) \preceq_{u\restriction B+B'} e' \preceq_{u\restriction B+B'} e$. Suppose to the contrary that there is $e' \in E_{u\restriction B+B'}$ such that $e' \neq e$ and $e' \neq \text{iccopy}_{u\restriction B+B'}(e)$ and $\text{iccopy}(e) \preceq_{u\restriction B+B'} e' \preceq_{u\restriction B+B'} e$. So, $\text{iccopy}(e) \preceq_u e' \preceq_u e$. We have the following cases.
  - If e is a B'-event, then $in_3^{-1}(e) \in E_s$ and $\text{iccopy}_{u\restriction B+B'}(e) = in_2(in_3^{-1}(e))$.
    * If e' is a B-event, then using (P4), we have $\text{iccopy}_{u\restriction B+B'}(e) \preceq_u e' \therefore in_2(in_3^{-1}(e)) \preceq_u e' \therefore in_2^{-1}(in_2(in_3^{-1}(e))) \preceq_s in_2^{-1}(e') \therefore in_3^{-1}(e) \preceq_s in_2^{-1}(e')$ and $e' \preceq_u e \therefore in_2^{-1}(e') \preceq_s in_3^{-1}(e)$. Since $\preceq_s$ is antisymmetric, we get $in_2^{-1}(e') = in_3^{-1}(e)$. We use (P5) to find that $in_2(in_3^{-1}(e)) \preceq_u in_2(in_2^{-1}(e')) \therefore \text{iccopy}_{u\restriction B+B'}(e) \preceq_u e'$ and $in_3(in_3^{-1}(e)) \preceq_u in_3(in_2^{-1}(e')) \therefore e \preceq_u e'$ and $in_2(in_2^{-1}(e')) \preceq_u in_2(in_3^{-1}(e)) \therefore e' \preceq_u \text{iccopy}_{u\restriction B+B'}(e)$ and $in_3(in_2^{-1}(e')) \preceq_u in_3(in_3^{-1}(e)) \therefore e' \preceq_u e$. We then use the fact that $\preceq_u$ is antisymmetric to find that $e = e' = \text{iccopy}_{u\restriction B+B'}(e)$ which contradicts our assumptions.
    * If e' is a B'-event, then using (P4), we have $\text{iccopy}_{u\restriction B+B'}(e) \preceq_u e' \therefore in_2(in_3^{-1}(e)) \preceq_u e' \therefore in_2^{-1}(in_2(in_3^{-1}(e))) \preceq_s in_3^{-1}(e') \therefore in_3^{-1}(e) \preceq_s in_3^{-1}(e')$ and $e' \preceq_u e \therefore in_3^{-1}(e') \preceq_s in_3^{-1}(e)$. Since $\preceq_s$ is antisymmetric, we get $in_3^{-1}(e') = in_3^{-1}(e)$. Since in_3 is injective, we conclude that $e = e'$ which contradicts our assumptions.
  - If e is a B-event, we use the same reasoning as the previous case.
- The fact that $u \restriction B+B'$ with $\text{iccopy}_{u\restriction B+B'}$ satisfies condition (C2) of Definition 5.36 is clear from the definition of λ_u.
- We verify that condition (C3) of Definition 5.36 holds.
  - Suppose $\lambda_{u\restriction B+B'}(e) \in I_B$ or $\lambda_{u\restriction B+B'}(\text{iccopy}_{u\restriction B+B'}(e)) \in I_{B'}$. It follows that $\lambda_s(in_2^{-1}(e)) \in I_B$. By (J4), $in_3(in_2^{-1}(e)) \frown_u in_2(in_2^{-1}(e))$. Therefore, $\text{iccopy}_{u\restriction B+B'}(e) \frown_u e$. Hence, $\text{iccopy}_{u\restriction B+B'}(e) \frown_{u\restriction B+B'} e$.
  - Suppose $\lambda_{u\restriction B+B'}(e) \notin I_B$ and $\lambda_{u\restriction B+B'}(\text{iccopy}_{u\restriction B+B'}(e)) \notin I_{B'}$. Let e be a B-event. By definition of λ_u, we have $\lambda_s(in_2^{-1}(e)) \notin I_B$. Since s is justified, there is $e' \in E_s$

such that $e' \curvearrowleft_s in_2^{-1}(e)$. Using (J3), we get $in_2(e') \curvearrowleft_u in_2(in_2^{-1}(e)) \therefore in_2(e') \curvearrowleft_u e \therefore in_2(e') \curvearrowleft_{u\restriction B+B'} e$ and $in_3(e') \curvearrowleft_u in_3(in_2^{-1}(e)) \therefore in_3(e') \curvearrowleft_u \text{iccopy}_{u\restriction B+B'}(e) \therefore in_3(e') \curvearrowleft_{u\restriction B+B'} \text{iccopy}_{u\restriction B+B'}(e)$. Since $in_3(e')$ is an B'-output in $u \restriction B+B'$, we get $\text{iccopy}_{u\restriction B+B'}(in_3(e')) = in_2(in_3^{-1}(in_3(e'))) = in_2(e')$. The case where e is a B'-event is similar.

We briefly argue that $u \restriction A+B = s$. We have $E_{u\restriction A+B} = E_{s\restriction A} + E_{s\restriction B}$. There is a bijection $\phi : E_s \to E_{u\restriction A+B}$ defined as

$$\phi(e) = \begin{cases} in_1(e) & \text{if } \lambda_s(e) \in inl(L_A) \\ in_2(e) & \text{if } \lambda_s(e) \in inr(L_B) \end{cases}$$

We need to prove that for all $e, e' \in E_s$, $e \preceq_s e'$ iff $\phi(e) \preceq_{u\restriction A+B} \phi(e')$. We have $e \preceq_s e' \Leftrightarrow in_j(e) \preceq_u in_k(e') \Leftrightarrow in_j(e) \preceq_{u\restriction A+B} in_k(e')$ where $j, k \in \{1, 2\}$.

The proof that $u \restriction A+B' = s$ is similar.

Next, we show the opposite inclusion, i.e. $\sigma; id_B \subseteq \sigma$. It is sufficient to prove that for any $u \in int_A(A, B, B')$

$$\text{if } u \restriction A+B = s \in \sigma \tag{5.1}$$

$$\text{and } u \restriction B+B' = t \in id_B \tag{5.2}$$

$$\text{then } u \restriction A+B' \in \sigma \tag{5.3}$$

We will show that $u \restriction A+B'$ is in the O, prefix and saturated closures of s. Suppose (5.1) and (5.2) are true. In the following, let us call inputs e in E_w such that $\text{ocopy}_w(e)$ is undefined *pending*.

1. Let u_1 be a trace that is like u, but where all pending B'-inputs occur at the end of u_1 in the same order they occur in u; that is,
   - $E_{u_1} = E_u$.
   - $\lambda_{u_1} = \lambda_u$.

- $\curvearrowleft_{u_1} = \curvearrowleft_u$.
- Define $P' = \{e \in E_u \mid \lambda_u(e) \in L_{B'}$ and $\pi \circ \lambda_t(e) = i$ and $\mathrm{ocopy}_t(e)$ is undefined$\}$. Let $\preceq_{u_1}$ be defined by the following, for all $e, e' \in E_u$.

  P1.1 If $e, e' \in E_u \setminus P'$, then $e \preceq_u e' \Leftrightarrow e \preceq_{u_1} e'$.

  P1.2 If $e, e' \in P'$, then $e \preceq_u e' \Leftrightarrow e \preceq_{u_1} e'$.

  P1.3 If $e \in E_u \setminus P'$ and $e' \in P'$, then $e \preceq_{u_1} e'$.

We can show that u_1 is an asynchronous trace; the proof is omitted. We show that $u \lesssim^{int} u_1$. First note that $u \upharpoonright B = u_1 \upharpoonright B$. Define the bijection $\phi : E_{u_1} \to E_u$ as the identity. We have by definition of u_1 that $\lambda_{u_1} = \lambda_u$ and $\curvearrowleft_{u_1} = \curvearrowleft_u$. For the last condition of Definition 5.46, let $e \preceq_{u_1} e'$.

- If [e is a B'-input or a B-input] and [e' is a B'-output or a B-output], then $e, e' \in E_{u_1} \setminus P'$ because e' is not a B'-input and if $e \in P'$, then $e \preceq_{u_1} e'$ implies $e' \in P'$ by (P1.2). Using (P1.1), we find that $e \preceq_u e'$.
- If [e is a B-output or an A-output] and [e' is a B-input or an A-input], then $e, e' \in E_{u_1} \setminus P'$. Using (P1.1), we find that $e \preceq_u e'$.

Note that $u_1 \upharpoonright A + B = s$. Applying Lemma 5.48 to $u \lesssim^{int} u_1$ we get $u \upharpoonright A + B' \lesssim u_1 \upharpoonright A + B'$. Let us call this result $(\star)$.

2. Let u_2 be a trace that is like u_1, but where all pending B'-inputs are removed. It is clear that $u_1 \leqslant^{int} u_2$ and $u_2 \upharpoonright A + B = s$. Applying Lemma 5.49 to $u_1 \leqslant^{int} u_2$ we get $u_1 \upharpoonright A + B' \leqslant u_2 \upharpoonright A + B'$. Let us call this result $(\blacktriangle)$.

3. Let u_3 be a trace that is like u_2, but where all pending B-inputs in u_2 have output copies concatenated at the end in the same order their input copies occur in u_2; that is, define $P = \{e \in E_{u_2} \mid \lambda_{u_2}(e) \in L_B$ and $\pi \circ \lambda_{u_2 \upharpoonright B+B'}(e) = i$ and $\mathrm{ocopy}_{u_2 \upharpoonright B+B'}(e)$ is undefined$\}$ and

   - $E_{u_3} = E_{u_2} + P$.
   - $\lambda_{u_3} = \lambda_{u_2} + \lambda_{u_2} \upharpoonright P$.
   - Let $\preceq_{u_3}$ be defined by the following, for all $e, e' \in E_{u_2}$.

P3.1 If $e, e' \in E_{u_2}$, then $e \preceq_{u_2} e' \Leftrightarrow in_1(e) \preceq_{u_3} in_1(e')$.

P3.2 If $e, e' \in P$, then $e \preceq_{u_2} e' \Leftrightarrow in_2(e) \preceq_{u_3} in_2(e')$.

P3.3 If $e \in E_{u_2}$ and $e' \in P$, then $in_1(e) \preceq_{u_3} in_2(e')$.

- Let $\curvearrowleft_{u_3}$ be defined by the following, for all $e, e' \in E_{u_2}$.

J3.1 If $e, e' \in E_{u_2}$, then $e \curvearrowleft_{u_2} e' \Leftrightarrow in_1(e) \curvearrowleft_{u_3} in_1(e')$.

J3.2 $e \in P$ and $e_0 \curvearrowleft_{u_2} e \Leftrightarrow in_1(\text{icopy}_{u_2 \restriction B+B'}(e_0)) \curvearrowleft_{u_3} in_2(e)$.

The proof that u_3 is an asynchronous trace is omitted. It is clear that $u_2 \preccurlyeq u_3$ and $u_3 \restriction A + B = s$. Note that $u_3 \restriction B + B'$ is of even length since each input has an output copy and vice versa. Applying Lemma 5.50 to $u_2 \preccurlyeq u_3$ we get $u_2 \restriction A + B' \preccurlyeq u_3 \restriction A + B'$. Let us call this result ($\bullet$).

4. Let u_4 be a trace that is like u_3, but where each B'-output directly follows its input copy in u_4 and each B'-input directly precedes its output copy in u_4; that is,

- $E_{u_4} = E_{u_3}$.
- $\lambda_{u_4} = \lambda_{u_3}$.
- $\curvearrowleft_{u_4} = \curvearrowleft_{u_3}$.
- Define copy $= \text{icopy}_{u_3 \restriction B+B'} \cup \text{ocopy}_{u_3 \restriction B+B'}$. We have copy is a well defined injection because the domains of $\text{icopy}_{u_3 \restriction B+B'}$ and $\text{ocopy}_{u_3 \restriction B+B'}$ are disjoint, and so are their codomains. Note that copy is an involution. The preorder $\preceq_{u_4}$ is defined as follows. For all $e, e' \in E_{u_3}$,

P4.1 if $e, e' \in E_{u_3 \restriction A}$, then $e \preceq_{u_3} e' \Leftrightarrow e \preceq_{u_4} e'$

P4.2 if $e \in E_{u_3 \restriction A}$ and $e' \in E_{u_3 \restriction B}$, then $[e \preceq_{u_3} e' \Leftrightarrow e \preceq_{u_4} e' \Leftrightarrow e \preceq_{u_4} \text{copy}(e')]$

P4.3 if $e \in E_{u_3 \restriction B}$ and $e' \in E_{u_3 \restriction A}$, then $[e \preceq_{u_3} e' \Leftrightarrow e \preceq_{u_4} e' \Leftrightarrow \text{copy}(e) \preceq_{u_4} e']$

P4.4 if $e, e' \in E_{u_3 \restriction B}$ and $e \neq e'$, then $[e \preceq_{u_3} e' \Leftrightarrow e \preceq_{u_4} e' \Leftrightarrow \text{copy}(e) \preceq_{u_4} \text{copy}(e') \Leftrightarrow \text{copy}(e) \preceq_{u_4} e' \Leftrightarrow e \preceq_{u_4} \text{copy}(e')]$

P4.5 $e \in E_{u_3 \restriction B}$ and $\pi \circ \lambda_{u_3 \restriction B+B'}(e) = i \Leftrightarrow e \preceq_{u_4} \text{copy}(e)$

P4.6 $e \in E_{u_3 \restriction B}$ and $\pi \circ \lambda_{u_3 \restriction B+B'}(e) = o \Leftrightarrow \text{copy}(e) \preceq_{u_4} e$.

We show that $(\forall e, e' \in E_{u_4})(e \curvearrowleft_{u_4} e' \Rightarrow e \preceq_{u_4} e')$ via a case study on the labels of e, e'.

- If $e, e' \in E_{u_4 \upharpoonright A}$, then $e \curvearrowleft_{u_4} e' \therefore e \curvearrowleft_{u_3} e' \therefore e \preceq_{u_3} e'$ as u_3 is well-formed $\therefore e \preceq_{u_4} e'$ by (P4.1).
- If $e, e' \in E_{u_4 \upharpoonright B}$, then $e \curvearrowleft_{u_4} e' \therefore e \curvearrowleft_{u_3} e' \therefore e \preceq_{u_3} e'$ as u_3 is well-formed $\therefore e \preceq_{u_4} e'$ by (P4.4).
- $e, e' \in E_{u_4 \upharpoonright B'}$, then $e \curvearrowleft_{u_4} e' \therefore e \curvearrowleft_{u_3} e'$. By Definition 5.37, $\text{copy}(e) \curvearrowleft_{u_3} \text{copy}(e') \therefore \text{copy}(e) \preceq_{u_3} \text{copy}(e')$. By (P4.4), $\text{copy}(\text{copy}(e)) \preceq_{u_4} \text{copy}(\text{copy}(e'))$. So, $e \preceq_{u_4} e'$ since copy is involutary.
- $e \in E_{u_4 \upharpoonright B}$ and $e' \in E_{u_4 \upharpoonright A}$, then $e \curvearrowleft_{u_4} e' \therefore e \curvearrowleft_{u_3} e' \therefore e \preceq_{u_3} e'$ as u_3 is well-formed $\therefore e \preceq_{u_4} e'$ by (P4.3).
- $e \in E_{u_4 \upharpoonright B'}$ and $e' \in E_{u_4 \upharpoonright B}$, then $e \curvearrowleft_{u_4} e' \therefore e \curvearrowleft_{u_3} e'$. By Definition 5.37, $e = \text{icopy}_{u_3 \upharpoonright B+B'}(e')$. We then use (P4.6) to find that $e \preceq_{u_4} e'$.
- All other label assignments are impossible.

The rest of the proof that u_4 is an asynchronous trace is omitted. We show that $u_3 \lesssim^{int} u_4$. Note that $u_3 \upharpoonright B = u_4 \upharpoonright B$. Define the bijection $\phi : E_{u_4} \to E_{u_3}$ as the identity. We have by definition of u_4 that $\lambda_{u_4} = \lambda_{u_3}$ and $\curvearrowleft_{u_4} = \curvearrowleft_{u_3}$. For the last condition of Definition 5.46, let $e \preceq_{u_4} e'$.

- If e is a B'-input and e' is a B'-output, then by (P4.4), $\text{copy}^{-1}(e) \preceq_{u_3} \text{copy}^{-1}(e')$. Since copy is involutary, $\text{copy}(e) \preceq_{u_3} \text{copy}(e')$. Since $u_3 \upharpoonright B + B' \in id_B$, we know that $\text{icopy}_{u_3 \upharpoonright B+B'}(\text{copy}(e)) \preceq_{u_3 \upharpoonright B+B'} \text{copy}(e)$ and $\text{copy}(e') \preceq_{u_3 \upharpoonright B+B'} \text{icopy}^{-1}_{u_3 \upharpoonright B+B'}(\text{copy}(e'))$. So, $e \preceq_{u_3} e'$.
- If e is a B'-input and e' is a B-output, then by (P4.4), $\text{copy}(e) \preceq_{u_3} e'$. We use the fact that $\text{icopy}_{u_3 \upharpoonright B+B'}(\text{copy}(e)) \preceq_{u_3} \text{copy}(e)$ to find that $e \preceq_{u_3} e'$.
- If e is a B-input and e' is a B'-output, then by (P4.4), $e \preceq_{u_3} \text{copy}(e')$. We use the fact that $\text{copy}(e') \preceq_{u_3 \upharpoonright B+B'} \text{icopy}^{-1}_{u_3 \upharpoonright B+B'}(\text{copy}(e'))$ to find that $e \preceq_{u_3} e'$.
- If e is a B-input and e' is a B-output, then by (P4.4), $e \preceq_{u_3} e'$.
- If [e is a B-output or an A-output] and [e' is a B-input or an A-input], then we use the same reasoning as the above four cases with (P4.1), (P4.2), (P4.3) or (P4.4) depending on the labels of e and e'.

Applying Lemma 5.48 to $u_3 \lesssim^{int} u_4$ we get $u_3 \upharpoonright A + B' \lesssim u_4 \upharpoonright A + B'$. Let us call this result ($\blacklozenge$). We also have $u_4 \upharpoonright B + B' \in cc_B$ because each input is directly followed by its output copy, and $u_4 \upharpoonright A + B = s$ from (P4.1)–(P4.3). Using Corollary 5.45, we get $u_4 \upharpoonright A + B' \lesssim s$. Let us call this result ($\blacktriangledown$).

Putting together ($\star$), ($\blacktriangle$), ($\bullet$), ($\blacklozenge$) and ($\blacktriangledown$) yields

$$u \upharpoonright A + B' \lesssim u_1 \upharpoonright A + B' \lessdot u_2 \upharpoonright A + B' \preccurlyeq u_3 \upharpoonright A + B' \lesssim u_4 \upharpoonright A + B' \lesssim s$$

Hence, $u \upharpoonright A + B' \in \sigma$.

The proof that $id_A; \sigma = \sigma$ is similar. ■

Lemma 5.52. *If $\sigma : A \to B$ and $\tau : C \to D$ are asynchronous processes, then $\sigma \otimes \tau$ is an asynchronous process.*

Proof. To prove that $\sigma \otimes \tau$ is prefix-closed, we use the same strategy as the proof of Lemma 5.15. For saturation, we follow the same strategy used in the proof of Lemma 5.39. Finally, the proof that $\sigma \otimes \tau$ is O-complete is similar to the proof in Lemma 5.40. ■

Lemma 5.53. *Tensor preserves the identity, i.e., $id_A \otimes id_B = id_{A \otimes B}$.*

The proof follows directly from Definition 5.37.

5.3 Round Abstraction on Causal Processes

In this section, we will show that partial round abstraction is compositional in $\mathbf{SynProc}_P$. We extend our definition of round abstraction to justified traces as follows.

Definition 5.54 (Round abstraction on justified traces)**.** *Let s and t be justified traces on A. We say that t* is a round abstraction of *s, written $s \sqsubseteq t$, if there exists a bijection $\phi : E_s \to E_t$ such that $\langle E_s, \lambda_s, \curvearrowleft_s \rangle$ and $\langle E_t, \lambda_t, \curvearrowleft_t \rangle$ are ϕ-isomorphic, i.e. $\lambda_s = \lambda_t \circ \phi$ and $\curvearrowleft_s = \curvearrowleft_t \circ \phi$, and ϕ is monotonic relative to temporal ordering, i.e. for any $e, e' \in E_s$, if $e \preceq_s e'$, then $\phi(e) \preceq_t \phi(e')$.*

We define partial round abstraction on causal and asynchronous processes exactly as we defined it on processes.

Definition 5.55 (Partial round abstraction). *For causal processes σ and τ over A, we say that τ is a* partial round abstraction *of σ, written $\sigma \sqsubseteq \tau$, if for any $t \in \tau$ there is $s \in \sigma$ such that $s \sqsubseteq t$.*

Partial round abstraction can be applied to causal and asynchronous processes compositionally.

Lemma 5.56. *Let $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$ be causal processes such that $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$. For all $u' \in \sigma' \between \tau'$, there are interaction traces $u, v \in int_J(A, B, C)$ and traces $s \in \sigma$ and $t \in \tau$ satisfying,*

1. *$u \sqsubseteq u'$ and $u \upharpoonright A + B = s$ and $u \upharpoonright B \in \Pi(t \upharpoonright B)$ and $u \upharpoonright C = t \upharpoonright C$*
2. *$v \sqsubseteq u'$ and $v \upharpoonright B + C = t$ and $v \upharpoonright B \in \Pi(s \upharpoonright B)$ and $v \upharpoonright A = s \upharpoonright A$*

The proof modifies the algorithm for constructing u and v by adding justification pointers. So we need to ensure, for each e justified by e', that e' occurs before or simultaneously with e. In all other respects, it is exactly like the proof of Lemma 4.10.

Proof. Let $u' \in \sigma' \between \tau'$; that is, $u' \upharpoonright A + B \in \sigma'$ and $u' \upharpoonright B + C \in \tau'$. Since $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$, it follows there are $s \in \sigma$ and $t \in \tau$ such that $s \sqsubseteq u' \upharpoonright A + B$ and $t \sqsubseteq u' \upharpoonright B + C$. So, let $\phi : E_s \to E_{u' \upharpoonright A+B}$ and $\psi : E_t \to E_{u' \upharpoonright B+C}$ be bijections such that $\lambda_s = \lambda_{u' \upharpoonright A+B} \circ \phi$ and $\lambda_t = \lambda_{u' \upharpoonright B+C} \circ \psi$. Without loss of generality, we will assume that ϕ and ψ are identities, i.e. that $E_s = E_{u' \upharpoonright A+B}$ and $E_t = E_{u' \upharpoonright B+C}$. It follows that $E_{s \upharpoonright B} = E_{t \upharpoonright B}$ and so $s \upharpoonright B$ is a permutation of $t \upharpoonright B$. The interaction trace u is defined as follows.

- $E_u = E_{u'}$
- $\lambda_u = \lambda_{u'}$
- $\curvearrowleft_u = \curvearrowleft_{u'}$
- We define $\preceq_u$ as follows. First, note that for any preorder $\preceq$, there is an associated strict partial order $\prec$, defined as $a \prec b$ if and only if $a \preceq b$ and $b \not\preceq a$. Let the relations $\preceq_1 \subseteq \preceq_{u'}$ and $\preceq_2 \subseteq \preceq_{u'}$ be defined as follows, for all $e, e' \in E_u$.

  P1 $e \prec_{u'} e' \Leftrightarrow e \preceq_1 e'$.

P2 If $e, e' \in E_s$, then $e \preceq_s e' \Leftrightarrow e \preceq_2 e'$.

P3 If $e, e' \in E_{t \upharpoonright C}$, then $e \preceq_t e' \Leftrightarrow e \preceq_2 e'$.

P4 If $e \in E_{t \upharpoonright C}$ and $e' \in E_{s \upharpoonright B}$, then $e \approx_{u'} e' \Leftrightarrow e \preceq_2 e'$.

P5 If $e \in E_{t \upharpoonright C}$ and $e' \in E_{s \upharpoonright A}$, then $e \approx_{u'} e' \Leftrightarrow e \preceq_2 e'$.

P6 If $e \in E_s$ and $e' \in E_{t \upharpoonright C}$, then $e \not\preceq_2 e'$.

We then set $\preceq_u = \preceq_1 \cup \preceq_2$.

We need to check that u is justified. Each non-initial is assigned a cause by $\curvearrowleft_u$ because u' is justified and $E_u = E_{u'}$ and $\curvearrowleft_u = \curvearrowleft_{u'}$. We still need to verify that $(\forall e, e' \in E_v)(e \curvearrowleft_u e' \Rightarrow e \preceq_u e')$. Suppose $e \curvearrowleft_u e'$. Then, $e \curvearrowleft_u e' \Leftrightarrow e \curvearrowleft_{u'} e' \Leftrightarrow e \preceq_{u'} e'$. There are two cases.

- $e \prec_{u'} e'$. By (P1), $e \preceq_u e'$.
- $e \approx_{u'} e'$. At this point, we have to consider the labels of e, e'.
  - If $\lambda_u(e) \in L_C$ and $\lambda_u(e) \in L_C$, then we have $e \curvearrowleft_{u'} e' \therefore e \curvearrowleft_{u' \upharpoonright B+C} e' \therefore e \curvearrowleft_t e'$ because $t \sqsubseteq u' \upharpoonright B + C$. So, $e \preceq_t e'$ because t is justified. By (P3), $e \preceq_u e'$.
  - If $\lambda_u(e) \in L_B$ and $\lambda_u(e) \in L_B$, or $\lambda_u(e) \in L_A$ and $\lambda_u(e) \in L_A$, or $\lambda_u(e) \in L_B$ and $\lambda_u(e) \in L_A$, then we have $e \curvearrowleft_{u'} e' \therefore e \curvearrowleft_{u' \upharpoonright A+B} e' \therefore e \curvearrowleft_s e'$ because $s \sqsubseteq u' \upharpoonright A + B$. So, $e \preceq_s e'$ because s is justified. By (P2), $e \preceq_u e'$.
  - If $\lambda_u(e) \in L_C$ and $\lambda_u(e) \in L_B$, then by (P4), $e \preceq_u e'$.

The rest of the proof is exactly the same as the proof of Lemma 4.10. The only addition is when we need to check that u satisfies the properties outlined above. For justification pointers, we use the following facts: $\curvearrowleft_u = \curvearrowleft_{u'}$ and $\curvearrowleft_{u \upharpoonright A+B} = \curvearrowleft_s$ and $\curvearrowleft_{u \upharpoonright C} = \curvearrowleft_{t \upharpoonright C}$. ■

We can now restate the main theorems.

Theorem 5.57 (Soundness I)**.** *For any causal processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\sigma \asymp \tau$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

Theorem 5.58 (Soundness II)**.** *For any causal processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\sigma' \bigcirc \tau'$, then $\sigma; \tau \sqsubseteq \sigma'; \tau'$.*

The proofs as exactly as given in Chapter 4.

Theorem 5.59 (Soundness III). *For any causal processes $\sigma, \sigma' : A \to B$ and $\tau, \tau' : B \to C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$, then $\sigma \otimes \tau \sqsubseteq \sigma' \otimes \tau'$.*

The proof is like the proof of Lemma 4.13, but we need to specify $\curvearrowleft_v$, which we define as $\curvearrowleft_s + \curvearrowleft_t$. The proof that v is justified and pointers are preserved between v and v' is omitted.

The above theorems also hold if σ and τ are asynchronous and σ' and τ' are causal. In this case, for Theorem 5.59, we additionally need to show that $\preceq_v$ is antisymmetric; the proof is omitted. For Theorem 5.57 and Theorem 5.58, we use the following variant of Lemma 5.56. Let $\Pi_A(s)$ denote the set of asynchronous permutations of s.

Lemma 5.60. *Let $\sigma : A \to B$, $\tau : B \to C$ be asynchronous processes and $\sigma' : A \to B$, $\tau' : B \to C$ be causal processes such that $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$. For all $u' \in \sigma' \between \tau'$, there are interaction traces $u, v \in int_A(A, B, C)$ and traces $s \in \sigma$ and $t \in \tau$ satisfying,*

1. *$u \sqsubseteq u'$ and $u \upharpoonright A + B = s$ and $u \upharpoonright B \in \Pi_A(t \upharpoonright B)$ and $u \upharpoonright C = t \upharpoonright C$*
2. *$v \sqsubseteq u'$ and $v \upharpoonright B + C = t$ and $v \upharpoonright B \in \Pi_A(s \upharpoonright B)$ and $v \upharpoonright A = s \upharpoonright A$*

Proof. Let $u' \in \sigma' \between \tau'$; that is, $u' \upharpoonright A + B \in \sigma'$ and $u' \upharpoonright B + C \in \tau'$. Since $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$, it follows there are $s \in \sigma$ and $t \in \tau$ such that $s \sqsubseteq u' \upharpoonright A + B$ and $t \sqsubseteq u' \upharpoonright B + C$. So, let $\phi : E_s \to E_{u' \upharpoonright A+B}$ and $\psi : E_t \to E_{u' \upharpoonright B+C}$ be bijections such that $\lambda_s = \lambda_{u' \upharpoonright A+B} \circ \phi$ and $\lambda_t = \lambda_{u' \upharpoonright B+C} \circ \psi$. Without loss of generality, we will assume that ϕ and ψ are identities, i.e. that $E_s = E_{u' \upharpoonright A+B}$ and $E_t = E_{u' \upharpoonright B+C}$. It follows that $E_{s \upharpoonright B} = E_{t \upharpoonright B}$ and so $s \upharpoonright B$ is a permutation of $t \upharpoonright B$. The interaction trace u is defined as in Lemma 5.56.

- $E_u = E_{u'}$
- $\lambda_u = \lambda_{u'}$
- $\curvearrowleft_u = \curvearrowleft_{u'}$
- We define $\preceq_u$ as follows. First, note that for any preorder $\preceq$, there is an associated strict partial order $\prec$, defined as $a \prec b$ if and only if $a \preceq b$ and $b \not\preceq a$. Let the relations $\preceq_1 \subseteq \preceq_{u'}$ and $\preceq_2 \subseteq \preceq_{u'}$ be defined as follows, for all $e, e' \in E_u$.

P1 $e \prec_{u'} e' \Leftrightarrow e \preceq_1 e'$.

P2 If $e, e' \in E_s$, then $e \preceq_s e' \Leftrightarrow e \preceq_2 e'$.

P3 If $e, e' \in E_{t \upharpoonright C}$, then $e \preceq_t e' \Leftrightarrow e \preceq_2 e'$.

P4 If $e \in E_{t \upharpoonright C}$ and $e' \in E_{s \upharpoonright B}$, then $e \approx_{u'} e' \Leftrightarrow e \preceq_2 e'$.

P5 If $e \in E_{t \upharpoonright C}$ and $e' \in E_{s \upharpoonright A}$, then $e \approx_{u'} e' \Leftrightarrow e \preceq_2 e'$.

P6 If $e \in E_s$ and $e' \in E_{t \upharpoonright C}$, then $e \not\preceq_2 e'$.

We then set $\preceq_u = \preceq_1 \cup \preceq_2$.

We need to check that $\preceq_u$ is antisymmetric, i.e. $(\forall e, e' \in E_u)(e \preceq_u e' \text{ and } e' \preceq_u e \Rightarrow e = e')$. Let $e \preceq_u e'$ and $e' \preceq_u e$, it follows from (P1) that $e \preceq_2 e'$ and $e' \preceq_2 e$. The proof is by case analysis on the labels of e and e'. By (P4) and (P5), cases (A, C), (B, C), (C, A), (C, B) are impossible. The remaining cases are (A, A), (A, B), (B, A), (B, B), (C, C). They are similar, so we only prove it for the case where $\lambda_v(e), \lambda_v(e') \in L_A$. We have $e \preceq_2 e' \Leftrightarrow e \preceq_s e'$ by (P2). Similarly, $e' \preceq_2 e \Leftrightarrow e' \preceq_s e$ by (P2). We get $e \preceq_s e'$ and $e' \preceq_s e$. So, $e = e'$ because $\preceq_s$ is antisymmetric.

The rest of the proof is exactly the same as the proof of Lemma 5.56. In particular, we proved that $u \upharpoonright B \in \Pi(t \upharpoonright B)$. Since u is an asynchronous trace, then $u \upharpoonright B$ is an asynchronous pre-trace and therefore $u \upharpoonright B \in \Pi_A(t \upharpoonright B)$. ■

5.4 Discussion

We built on Chapter 3 to introduce justification pointers to synchronous processes thereby defining $\mathbf{SynProc}_P$, a category of causal synchronous processes. Justification pointers allow us to encode causality in traces in a way that is preserved by the various operations of the category—in particular, the tensor product.

Alternatively, instead of explicit justification pointers, we may assign causality according to rules or principles that take into account the structure of the trace. For example, the 'serial causation' principle used in pointer-free game models [GM03, GS10] assigns, to each event, the most recent occurrence of an event that can cause it as its justifier. However, it is a significant challenge to formulate such principles so that they are stable with respect to various trace operations, in particular, interleaving.

We then described **AsyProc**, a category of asynchronous processes. Structurally, this category is equivalent to the full subcategory of $\mathcal{G}$—the category of multi-threaded saturated strategies [GM08, pp. 14–18]—whose objects consist of questions only. We introduced new formalisations of the saturation preorder (Definition 5.31 and Definition 5.32), alternating copycat (Definition 5.36) and asynchronous identity (Definition 5.37); and presented new categorical proofs.

Next, we demonstrated that partial round abstraction is compositional on processes in either category. So, we may apply partial round abstraction to reduce the latency of processes within $\mathbf{SynProc}_p$. Moreover, when applied to processes in **AsyProc**, partial round abstraction yields locally synchronous processes. This is one of the main contributions of this thesis: a technique that allows deriving synchronous representations of asynchronous processes compositionally. Since justification pointers are only needed to define saturated processes, our results transpose without difficulty to any models where causality may be unambiguously determined—for example, models of hardware circuits [GS10].

Further, our results are of particular interest to hardware synthesis using GoS [Ghi07] because they permit deriving lower latency synchronous processes from game semantic strategies. Nevertheless, partial round abstraction allows traces from the original process to have no corresponding traces in the abstraction. Hence, in order to obtain synchronous processes that most accurately represent the original asynchronous semantics, we require a *total* form of round abstraction, as described in Section 4.4. The results of this chapter may therefore be seen as a stepping stone towards a more comprehensive form of compositional round abstraction.

CHAPTER 6

GLOBAL CLOCKS AND DETERMINISM

In this chapter, we show that the locally synchronous framework of Chapter 3 can be wired with a global clock. To this end, we demonstrate that locally-synchronous processes can be lifted to global synchrony in a principled way. As a consequence, the results of Chapter 4 can be extended to systems using global clocks, the predominant digital design paradigm.

We will also study deterministic processes. Determinism is a desirable and sometimes necessary feature of real-time systems. In particular, *reactive* systems [HP85, Hal93, BB91] are often required to guarantee the same behaviour for each input. For this reason, determinism is an essential facet of synchronous languages.

In Section 6.1, we introduce the notions of *clock monoid* and *clock monad*, which are constructions that allow us to extend processes with global clocks. Next, in Section 6.2, we describe a category of processes that synchronise with a global clock. We then study deterministic processes in Section 6.3. We finish with a discussion.

6.1 Clock Monad and Clock Monoid*

Monads are category theoretic constructions [Mac98] that have been used to model computational effects since Moggi [Mog89, Mog91]. In this section, we follows Moggi's methodology to extend our model of processes with a global clock. In particular, we use a *strong monad* [Koc70, Koc72], which has a natural transformation that relates tensor product and monad.

Let us first recall a few definitions.

*Extended version of [GM10, Section 2.1]

Definition 6.1 (Monad). *A monad on a category* $\mathbf{C}$ *is a triple* $\langle T, \eta, \mu \rangle$*, where* T *is an endofunctor in* $\mathbf{C}$*,* $\eta : I \to T$*,* $\mu : T^2 \to T$ *are natural transformations, and the following diagrams commute.*

$$\begin{array}{ccc} T^3A & \xrightarrow{\mu_{TA}} & T^2A \\ {\scriptstyle T\mu_A}\downarrow & & \downarrow{\scriptstyle \mu_A} \\ T^2A & \xrightarrow[\mu_A]{} & TA \end{array} \qquad \begin{array}{ccccc} TA & \xrightarrow{\eta_{TA}} & T^2A & \xleftarrow{T\eta_A} & TA \\ & {\scriptstyle id_{TA}}\searrow & \downarrow{\scriptstyle \mu_A} & \swarrow{\scriptstyle id_{TA}} & \\ & & TA & & \end{array}$$

Definition 6.2 (Strong monad). *A strong monad on a symmetric monoidal category* $\mathbf{C}$ *is a monad,* $\langle T, \eta, \mu \rangle$ *equipped with a* tensorial strength *natural transformation* t *that associates to each pair of objects* A, B*, a morphism* $t_{A,B} : A \otimes TB \to T(A \otimes B)$ *such that the following diagrams commute.*

$$\begin{array}{ccccc} (A \otimes B) \otimes TC & \xrightarrow{t_{A\otimes B,C}} & T((A \otimes B) \otimes C) & & \\ {\scriptstyle \alpha_{A,B,TC}}\downarrow & & & \searrow{\scriptstyle T\alpha_{A,B,C}} & \\ A \otimes (B \otimes TC) & \xrightarrow[id_A \otimes t_{B,C}]{} & A \otimes T(B \otimes C) & \xrightarrow[t_{A,B\otimes C}]{} & T(A \otimes (B \otimes C)) \end{array} \qquad \begin{array}{ccc} I \otimes TA & \xrightarrow{t_{I,A}} & T(I \otimes A) \\ & {\scriptstyle \lambda_{TA}}\searrow & \downarrow{\scriptstyle T\lambda_A} \\ & & TA \end{array}$$

$$\begin{array}{ccccc} A \otimes T^2B & \xrightarrow{id_A \otimes \mu_B} & A \otimes TB & \xleftarrow{id_A \otimes \eta_B} & A \otimes B \\ {\scriptstyle t_{A,TB}}\downarrow & & & \searrow{\scriptstyle t_{A,B}} & \downarrow{\scriptstyle \eta_{A\otimes B}} \\ T(A \otimes TB) & \xrightarrow[Tt_{A,B}]{} & T^2(A \otimes B) & \xrightarrow[\mu_{A\otimes B}]{} & T(A \otimes B) \end{array}$$

The strength of a strong monad $\langle T, \eta, \mu, t \rangle$ on a symmetric monoidal category $\mathbf{C}$ gives rise to a *costrength* natural transformation t' that associates to each pair of objects A and B, a morphism $t'_{A,B} : TA \otimes B \to T(A \otimes B)$ defined as follows.

$$t'_{A,B} : TA \otimes B \xrightarrow{\gamma_{TA,B}} B \otimes TA \xrightarrow{t_{B,A}} T(B \otimes A) \xrightarrow{T\gamma_{B,A}} T(A \otimes B)$$

A strong monad is *commutative* when the following diagram commutes.

$$\begin{array}{ccccccc} & & T(TA \otimes B) & \xrightarrow{Tt'_{A,B}} & T^2(A \otimes B) & & \\ & {\scriptstyle t_{TA,B}}\nearrow & & & & \searrow{\scriptstyle \mu_{A\otimes B}} & \\ TA \otimes TB & & & & & & T(A \otimes B) \\ & {\scriptstyle t'_{A,TB}}\searrow & & & & \nearrow{\scriptstyle \mu_{A\otimes B}} & \\ & & T(A \otimes TB) & \xrightarrow[Tt_{A,B}]{} & T^2(A \otimes B) & & \end{array}$$

Definition 6.3 (Monoid)**.** *A monoid $\langle M, \eta, \mu \rangle$ in a monoidal category* $\mathbf{C}$ *is an object M together with two morphisms $\eta : I \to M$ and $\mu : M \otimes M \to M$ such that the following diagrams commute.*

$$\begin{array}{ccccc} (M \otimes M) \otimes M & \xrightarrow{\alpha_{M,M,M}} & M \otimes (M \otimes M) & \xrightarrow{id_M \otimes \mu} & M \otimes M \\ \downarrow{\scriptstyle \mu \otimes id_M} & & & & \downarrow{\scriptstyle \mu} \\ M \otimes M & & \xrightarrow[\mu]{\hspace{6em}} & & M \end{array}$$

$$\begin{array}{ccccc} I \otimes M & \xrightarrow{\eta \otimes id_M} & M \otimes M & \xleftarrow{id_M \otimes \eta} & M \otimes I \\ & {\scriptstyle \lambda_M} \searrow & \downarrow{\scriptstyle \mu} & \swarrow {\scriptstyle \rho_M} & \\ & & M & & \end{array}$$

If $\mathbf{C}$ *has a symmetry γ and $\gamma; \mu = \mu$, then the monoid is* commutative.

We can now define our *clock monad*. Let Ck be a reserved one-port object in $\mathbf{SynProc}_P$ with a single label *tick*. The functor $T : \mathbf{SynProc}_P \to \mathbf{SynProc}_P$ is defined as follows.

$$\text{On objects: } T(A) = A \otimes Ck,$$
$$\text{On morphisms: } T(f) = f \otimes id_{Ck}$$

We also define natural transformation (at object A) $\eta_A : A \to T(A)$ as,

$$\eta_A = \{s \in \Theta_J(A \Rightarrow A' \otimes Ck) \mid s \upharpoonright (A + A') \in id_A\}$$

and natural transformation (at object A) $\mu_A : T^2(A) \to T(A)$ as,

$$\mu_A = \{s \in \Theta_J(((A \otimes Ck) \otimes Ck') \Rightarrow A' \otimes Ck'') \mid$$
$$s \upharpoonright (A + A') \in id_A \text{ and } s \upharpoonright (Ck + Ck'') \in id_{Ck} \text{ and } s \upharpoonright (Ck' + Ck'') \in id_{Ck}\}.$$

The clock monad is a specification of how processes equipped with a clock should be 'wired'. It does not describe nor enforce how the processes use the clock.

Applying T to processes interleaves them with the behaviour of the clock. The effect of η_A and μ_A is to remove and to duplicate the clock behaviour, respectively. These constructions are illustrated by the circuit-like diagrams in Figure 6.1. Signature A has an arbitrary number of

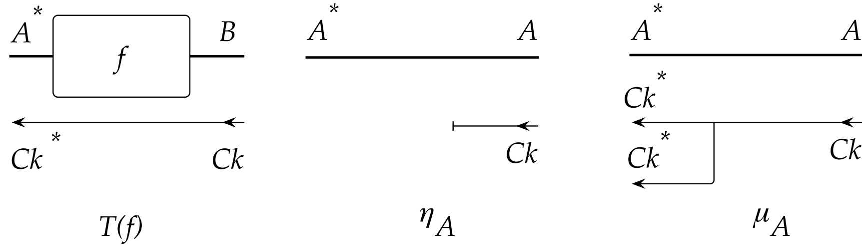


Figure 6.1: Clock monad

ports, but Ck is single-port; we show its input/output polarity with arrows for extra clarity.

The following are alternative characterisations. We have $\eta_A = \rho_A^{-1}; id_A \otimes cs_{Ck}$ where $cs_{Ck} : I \to Ck$ is the set of traces consisting of *tick* occurrences, i.e. $tick^*$. Similarly, $\mu_A = \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck}$ where $sp_{Ck} : Ck \otimes Ck \to Ck$ is the set of traces defined by $\{s \in \Theta_J(Ck \otimes Ck' \Rightarrow Ck'') \mid s \restriction (Ck + Ck'') \in id_{Ck}$ and $s \restriction (Ck' + Ck'') \in id_{Ck}\}$. A direct informal characterisation is $\langle tick, tick', tick'' \rangle^*$ where $tick \in L_{Ck}$, $tick' \in L_{Ck'}$ and $tick'' \in L_{Ck''}$.

We first show that $\langle Ck, sp_{Ck}, cs_{Ck} \rangle$ is a monoid in $\mathbf{SynProc}_P$.

Lemma 6.4. *The triple $\langle Ck, sp_{Ck}, cs_{Ck} \rangle$ is a monoid in $\mathbf{SynProc}_P$.*

Proof. We have to show the following equalities.

$$\alpha_{Ck,Ck,Ck}; id_{Ck} \otimes sp_{Ck}; sp_{Ck} = sp_{Ck} \otimes id_{Ck}; sp_{Ck}. \tag{6.1}$$

$$\alpha^{-1}_{Ck,Ck,Ck}; sp_{Ck} \otimes id_{Ck}; sp_{Ck} = id_{Ck} \otimes sp_{Ck}; sp_{Ck}. \tag{6.2}$$

$$cs_{Ck} \otimes id_{Ck}; sp_{Ck} = \lambda_{Ck}. \tag{6.3}$$

$$id_{Ck} \otimes cs_{Ck}; sp_{Ck} = \rho_{Ck}. \tag{6.4}$$

We begin by proving (6.1). Expanding the definitions, we have,

$$\begin{aligned}
LHS = \{u \in \Theta_J(((((Ck_4 \otimes CK_5) \otimes Ck_6) \Rightarrow (Ck'_4 \otimes (CK'_5 \otimes Ck'_6))) \Rightarrow Ck_2 \otimes Ck_3) \Rightarrow Ck_1) \mid \\
u \restriction Ck_2 + Ck_1 \in id_{Ck} \text{ and } u \restriction Ck_3 + Ck_1 \in id_{Ck} \text{ and } u \restriction Ck'_4 + Ck_2 \in id_{Ck} \text{ and} \\
u \restriction Ck'_5 + Ck_3 \in id_{Ck} \text{ and } u \restriction Ck'_6 + Ck_3 \in id_{Ck} \text{ and } u \restriction Ck_4 + Ck'_4 \in id_{Ck} \text{ and} \\
u \restriction Ck_5 + Ck'_5 \in id_{Ck} \text{ and } u \restriction Ck_6 + Ck'_6 \in id_{Ck}\} \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1 \\
RHS = \{v \in \Theta_J((((Ck_4 \otimes CK_5) \otimes Ck_6) \Rightarrow Ck_2 \otimes Ck_3) \Rightarrow Ck_1) \mid \\
v \restriction Ck_2 + Ck_1 \in id_{Ck} \text{ and } v \restriction Ck_3 + Ck_1 \in id_{Ck} \text{ and } v \restriction Ck_4 + Ck_2 \in id_{Ck} \text{ and}
\end{aligned}$$

$$v \upharpoonright Ck_5 + Ck_2 \in id_{Ck} \text{ and } v \upharpoonright Ck_6 + Ck_3 \in id_{Ck}\} \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_1$$

We first show that $\alpha_{Ck,Ck,Ck}; id_{Ck} \otimes sp_{Ck}; sp_{Ck} \subseteq sp_{Ck} \otimes id_{Ck}; sp_{Ck}$.

Let us recall some notation introduced in Chapter 3. For $u \in id_A$ and events $e, e' \in E_u$, where $u \in id_A$, let us write $e \leftrightharpoons_u e'$ when $e \approx_u e'$ and $[\lambda_u(e) = inl(a)$ if and only if $\lambda_u(e') = inr(a)]$ and $[\lambda_u(e) = inr(a)$ if and only if $\lambda_u(e') = inl(a)]$ where $a \in L_A$.

Let $u \in \alpha_{Ck,Ck,Ck} \fatsemi id_{Ck} \otimes sp_{Ck} \fatsemi sp_{Ck}$. We find a trace $v \in sp_{Ck} \otimes id_{Ck} \fatsemi sp_{Ck}$ such that $u \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_1 = v \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_1$. We define v as follows.

- $E_v = E_{u \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}$
- $\lambda_v = \lambda_{u \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}$
- $\preceq_v = \preceq_{u \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}$
- $\curvearrowleft_v$ is defined as follows, for all $e, e' \in E_v$.

  JV1 If $e, e' \in E_{u \upharpoonright Ck_2 + Ck_3 + Ck_1}$, then $e \curvearrowleft_u e' \Leftrightarrow e \curvearrowleft_v e'$.

  JV2 If $e \in E_{u \upharpoonright Ck_2}$ and $e' \in E_{u \upharpoonright Ck_4}$, then $(\exists e'' \in E_{u \upharpoonright Ck'_4})(e \curvearrowleft_u e''$ and $e'' \curvearrowleft_u e') \Leftrightarrow e \curvearrowleft_v e'$.

  JV3 If $e \in E_{u \upharpoonright Ck_2}$ and $e' \in E_{u \upharpoonright Ck_5}$, then $(\exists e_5 \in E_{u \upharpoonright Ck'_5}, \exists e_3 \in E_{u \upharpoonright Ck_3}, \exists e_1 \in E_{u \upharpoonright Ck_1})(e_1 \curvearrowleft_u e$ and $e_1 \curvearrowleft_u e_3$ and $e_3 \curvearrowleft_u e_5$ and $e_5 \curvearrowleft_u e') \Leftrightarrow e \curvearrowleft_v e'$.

  JV4 If $e \in E_{u \upharpoonright Ck_3}$ and $e' \in E_{u \upharpoonright Ck_6}$, then $(\exists e'' \in E_{u \upharpoonright Ck'_6})(e \curvearrowleft_u e''$ and $e'' \curvearrowleft_u e') \Leftrightarrow e \curvearrowleft_v e'$.

  JV5 The only pointers in $\curvearrowleft_v$ are those specified by (JV1)–(JV4); that is, if $[e \in E_{u \upharpoonright Ck_1}$ and $e' \in E_{u \upharpoonright Ck_4}]$ or $[e \in E_{u \upharpoonright Ck_1}$ and $e' \in E_{u \upharpoonright Ck_5}]$ or $[e \in E_{u \upharpoonright Ck_1}$ and $e' \in E_{u \upharpoonright Ck_6}]$ or $[e \in E_{u \upharpoonright Ck_2}$ and $e' \in E_{u \upharpoonright Ck_6}]$ or $[e \in E_{u \upharpoonright Ck_3}$ and $e' \in E_{u \upharpoonright Ck_4}]$ or $[e \in E_{u \upharpoonright Ck_3}$ and $e' \in E_{u \upharpoonright Ck_5}]$ or $[e \in E_{u \upharpoonright Ck_4}$ and $e' \in E_{u \upharpoonright Ck_1}]$ or $[e \in E_{u \upharpoonright Ck_4}$ and $e' \in E_{u \upharpoonright Ck_2}]$ or $[e \in E_{u \upharpoonright Ck_4}$ and $e' \in E_{u \upharpoonright Ck_3}]$ or $[e \in E_{u \upharpoonright Ck_4}$ and $e' \in E_{u \upharpoonright Ck_4}]$ or $[e \in E_{u \upharpoonright Ck_4}$ and $e' \in E_{u \upharpoonright Ck_5}]$ or $[e \in E_{u \upharpoonright Ck_4}$ and $e' \in E_{u \upharpoonright Ck_6}]$ or $[e \in E_{u \upharpoonright Ck_5}$ and $e' \in E_{u \upharpoonright Ck_1}]$ or $[e \in E_{u \upharpoonright Ck_5}$ and $e' \in E_{u \upharpoonright Ck_2}]$ or $[e \in E_{u \upharpoonright Ck_5}$ and $e' \in E_{u \upharpoonright Ck_3}]$ or $[e \in E_{u \upharpoonright Ck_5}$ and $e' \in E_{u \upharpoonright Ck_4}]$ or $[e \in E_{u \upharpoonright Ck_5}$ and $e' \in E_{u \upharpoonright Ck_5}]$ or $[e \in E_{u \upharpoonright Ck_5}$ and $e' \in E_{u \upharpoonright Ck_6}]$ or $[e \in E_{u \upharpoonright Ck_6}$ and $e' \in E_{u \upharpoonright Ck_1}]$ or $[e \in E_{u \upharpoonright Ck_6}$ and $e' \in E_{u \upharpoonright Ck_2}]$ or $[e \in E_{u \upharpoonright Ck_6}$ and $e' \in E_{u \upharpoonright Ck_3}]$ or $[e \in E_{u \upharpoonright Ck_6}$ and $e' \in E_{u \upharpoonright Ck_4}]$ or $[e \in E_{u \upharpoonright Ck_6}$ and $e' \in E_{u \upharpoonright Ck_5}]$ or $[e \in E_{u \upharpoonright Ck_6}$ and $e' \in E_{u \upharpoonright Ck_6}]$, then $e \not\curvearrowleft_v e'$.

We show that v is justified, i.e. that $\curvearrowleft_v\colon E_v \setminus \{e \in E_v \mid \lambda_v(e) \in I_{Ck_1}\} \to E_v$ is total and $(\forall e, e' \in E_v)(e \curvearrowleft_v e' \Rightarrow (e \preceq_v e \text{ and } \lambda_v(e) \vdash \lambda_v(e'))$. By inspection of the definition, we can see that the partial function $\curvearrowleft_v$ is total because every non-initial event has a pointer. In particular, for any $e' \in E_{u \restriction Ck_5}$, the existence of events e, e_5, e_3, e_1 in the definition is guaranteed since $u \restriction Ck_2 + Ck_1 \in id_{Ck}$ and $u \restriction Ck_3 + Ck_1 \in id_{Ck}$ and $u \restriction Ck'_5 + Ck_3 \in id_{Ck}$ and $u \restriction Ck_5 + Ck'_5 \in id_{Ck}$.

Now, we show $(\forall e, e' \in E_v)(e \curvearrowleft_v e' \Rightarrow (e \preceq_v e \text{ and } \lambda_v(e) \vdash \lambda_v(e'))$. Suppose $e \curvearrowleft_v e'$. By definition of $\curvearrowleft_v$, we have the following cases.

- If $e, e' \in E_{u \restriction Ck_2 + Ck_3 + Ck_1}$, then $e \curvearrowleft_v e' \therefore e \curvearrowleft_u e' \therefore e \preceq_u e' \therefore e \preceq_v e'$.
- If $e \in E_{u \restriction Ck_2}$ and $e' \in E_{u \restriction Ck_4}$, then $e \curvearrowleft_v e' \therefore (\exists e'' \in E_{u \restriction Ck'_4})(e \curvearrowleft_u e'' \text{ and } e'' \curvearrowleft_u e') \therefore (\exists e'' \in E_{u \restriction Ck'_4})(e \preceq_u e'' \text{ and } e'' \preceq_u e') \therefore e \preceq_u e' \therefore e \preceq_v e'$.
- If $e \in E_{u \restriction Ck_2}$ and $e' \in E_{u \restriction Ck_5}$, then $e \curvearrowleft_v e' \therefore (\exists e_5 \in E_{u \restriction Ck'_5}, \exists e_3 \in E_{u \restriction Ck_3}, \exists e_1 \in E_{u \restriction Ck_1})(e_1 \curvearrowleft_u e \text{ and } e_1 \curvearrowleft_u e_3 \text{ and } e_3 \curvearrowleft_u e_5 \text{ and } e_5 \curvearrowleft_u e')$. Since Ck is a single-event signature, in each trace $s \in id_{Ck}$, we have by the definition of id that $e \curvearrowleft_s e'$ implies $e \approx_s e'$. As $u \restriction Ck_2 + Ck_1 \in id_{Ck}$ and $u \restriction Ck_3 + Ck_1 \in id_{Ck}$ and $u \restriction Ck'_5 + Ck_3 \in id_{Ck}$ and $u \restriction Ck_5 + Ck'_5 \in id_{Ck}$, we get $e \approx_u e_1 \approx_u e_3 \approx_u e_5 \approx_u e' \therefore e \preceq_v e'$.
- If $e \in E_{u \restriction Ck_3}$ and $e' \in E_{u \restriction Ck_6}$, then $(\exists e'' \in E_{u \restriction Ck'_6})(e \curvearrowleft_u e'' \text{ and } e'' \curvearrowleft_u e') \therefore (\exists e'' \in E_{u \restriction Ck'_6})(e \preceq_u e'' \text{ and } e'' \preceq_u e') \therefore e \preceq_u e' \therefore e \preceq_v e'$.

Note that in each case $\lambda_v(e) \vdash \lambda_v(e')$.

If follows from its definition that v satisfies $v \restriction Ck_2 + Ck_1 \in id_{Ck}$ and $v \restriction Ck_3 + Ck_1 \in id_{Ck}$. The fact that $v \restriction Ck_4 + Ck_2 \in id_{Ck}$ follows from $u \restriction Ck'_4 + Ck_2 \in id_{Ck}$ and $u \restriction Ck_4 + Ck'_4 \in id_{Ck}$ and the definition of $\curvearrowleft_v$. The fact that $v \restriction Ck_6 + Ck_3 \in id_{Ck}$ follows from $u \restriction Ck'_6 + Ck_3 \in id_{Ck}$ and $u \restriction Ck_6 + Ck'_6 \in id_{Ck}$ and the definition of $\curvearrowleft_v$.

We still have to show $v \restriction Ck_5 + Ck_2 \in id_{Ck}$. Let $e_5 \in E_{u \restriction Ck_5}$. We have the following facts.

- Since $u \restriction Ck_5 + Ck'_5 \in id_{Ck}$, there is $e'_5 \in E_{u \restriction Ck'_5}$ such that $e_5 \leftrightharpoons_{u \restriction Ck_5 + Ck'_5} e'_5$.
- As $u \restriction Ck'_5 + Ck_3 \in id_{Ck}$, there is $e_3 \in E_{u \restriction Ck_3}$ such that $e'_5 \leftrightharpoons_{u \restriction Ck'_5 + Ck_3} e_3$.
- Because $u \restriction Ck_3 + Ck_1 \in id_{Ck}$, there is $e_1 \in E_{u \restriction Ck_1}$ such that $e_3 \leftrightharpoons_{u \restriction Ck_3 + Ck_1} e_1$.

- Since $u \restriction Ck_2 + Ck_1 \in id_{Ck}$, there is $e_2 \in E_{u \restriction Ck_2}$ such that $e_1 \leftrightharpoons_{u \restriction Ck_2 + Ck_1} e_2$.

Since $\preceq_u$ is transitive, $e_1 \approx_u e_2 \approx_u e_3 \approx_u e_5$. So, for all $e_5 \in E_{u \restriction Ck_5}$ there is $e_2 \in E_{u \restriction Ck_2}$ such that $e_5 \leftrightharpoons_{u \restriction Ck_5 + Ck_2} e_2$. We use the same reasoning to conclude that for all $e_2 \in E_{u \restriction Ck_2}$ there is $e_5 \in E_{u \restriction Ck_5}$ such that $e_5 \leftrightharpoons_{u \restriction Ck_5 + Ck_2} e_2$. In essence, for all $e \in E_{u \restriction Ck_5 + Ck_2}$, there is $e' \in E_{u \restriction Ck_5 + Ck_2}$ such that $e \leftrightharpoons_{u \restriction Ck_5 + Ck_2} e'$. Since $E_v = E_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}$ and $\lambda_v = \lambda_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}$ and $\preceq_v = \preceq_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}$, we have for all $e \in E_{v \restriction Ck_5 + Ck_2}$, there is $e' \in E_{v \restriction Ck_5 + Ck_2}$ such that $e \leftrightharpoons_{v \restriction Ck_5 + Ck_2} e'$.

We need to verify that $\curvearrowleft_{v \restriction Ck_5 + Ck_2}$ conforms to Definition 5.11. This can be noted by inspecting the definition of $\curvearrowleft_v$.

We now show that $u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1 = v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1$. From the definition of v and the definition of projection, $E_{v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} = E_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1}$, $\lambda_{v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} = \lambda_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1}$ and $\preceq_{v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} = \preceq_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1}$. Next, we prove that for all $e, e' \in E_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1}$, we have $e \curvearrowleft_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \Leftrightarrow e \curvearrowleft_{v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} e'$ through a case study on the labels of e, e'.

- Case: $\lambda_u(e) \in L_{Ck_1}$ and $\lambda_u(e') \in L_{Ck_4}$. This is similar to the following case. We use (JV1) and (JV2).

- Case: $\lambda_u(e) \in L_{Ck_1}$ and $\lambda_u(e') \in L_{Ck_5}$. Let $e \curvearrowleft_{v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} e'$. So, there is $e_2 \in E_{u \restriction Ck_2}$ such that $e \curvearrowleft_v e_2$ and $e_2 \curvearrowleft_v e'$. Using (JV3) on $e_2 \curvearrowleft_v e'$, there is $e_1 \in E_{u \restriction Ck_1}$ such that $e_1 \curvearrowleft_u e_2$ and $e_1 \curvearrowleft_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} e'$. Using (JV1) on $e \curvearrowleft_v e_2$, we get $e \curvearrowleft_u e_2$. Therefore, $e_1 = e$, so $e \curvearrowleft_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} e'$. For the opposite direction, let $e \curvearrowleft_{u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} e'$. So, there are $e_3 \in E_{u \restriction Ck_3}, e_5 \in E_{u \restriction Ck'_5}$ such that $e \curvearrowleft_u e_3$ and $e_3 \curvearrowleft_u e_5$ and $e_5 \curvearrowleft_u e'$. Since $u \restriction Ck_2 + Ck_1 \in id_{Ck}$, there is $e_2 \in E_{u \restriction Ck_2}$ such that $e \curvearrowleft_u e_2$. By (JV1), $e \curvearrowleft_v e_2$. By (JV3), $e_2 \curvearrowleft_v e'$. So, $e \curvearrowleft_{v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1} e'$.

- Case: $\lambda_u(e) \in L_{Ck_1}$ and $\lambda_u(e') \in L_{Ck_6}$. This is similar to the previous case. We use (JV1) and (JV4).

- All other label assignments are illegal.

Next, we show that $\alpha_{Ck,Ck,Ck}; id_{Ck} \otimes sp_{Ck}; sp_{Ck} \supseteq sp_{Ck} \otimes id_{Ck}; sp_{Ck}$.

Let $v \in sp_{Ck} \otimes id_{Ck} \lightning sp_{Ck}$. We find a trace $u \in \alpha_{Ck,Ck,Ck} \lightning id_{Ck} \otimes sp_{Ck} \lightning sp_{Ck}$ such that $v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1 = u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1$. We define u using the same algorithm described in the proof of Lemma 3.24.

- $E_u = E_{v \restriction Ck_1+Ck_2+Ck_3+Ck_4+Ck_5+Ck_6} + E_{v \restriction Ck_4+Ck_5+Ck_6}$
- $\lambda_u = \lambda_{v \restriction Ck_1+Ck_2+Ck_3+Ck_4+Ck_5+Ck_6} + \lambda_{v \restriction Ck_4+Ck_5+Ck_6}$
- $\preceq_u$ is defined as follows for all $e, e' \in E_v$.

  P1 If $e, e' \in E_{v \restriction Ck_1+Ck_2+Ck_3}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e')$

  P2 If $e \in E_{v \restriction Ck_1+Ck_2+Ck_3}$ and $e' \in E_{v \restriction Ck_4+Ck_5+Ck_6}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_2(e')$

  P3 If $e \in E_{v \restriction Ck_4+Ck_5+Ck_6}$ and $e' \in E_{v \restriction Ck_1+Ck_2+Ck_3}$, then $e \preceq_v e' \Leftrightarrow in_2(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_1(e')$

  P4 If $e, e' \in E_{v \restriction Ck_4+Ck_5+Ck_6}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_2(e) \preceq_u in_2(e') \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_2(e) \preceq_u in_1(e')$

- $\curvearrowleft_u$ is defined as follows, for all $e, e' \in E_v$.

  J1 If $e, e' \in E_{v \restriction Ck_1+Ck_2+Ck_3}$, then $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_u in_1(e')$.

  J2 If $e \in E_{v \restriction Ck_2}$ and $e' \in E_{v \restriction Ck_4}$, then $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_u in_2(e') \Leftrightarrow in_2(e') \curvearrowleft_u in_1(e')$.

  J3 If $e \in E_{v \restriction Ck_3}$ and $e' \in E_{v \restriction Ck_5}$, then $(\exists e_1 \in E_{v \restriction Ck_1}, \exists e_2 \in E_{v \restriction Ck_2})(e_1 \curvearrowleft_v e$ and $e_1 \curvearrowleft_v e_2$ and $e_2 \curvearrowleft_v e') \Leftrightarrow in_1(e) \curvearrowleft_u in_2(e') \Leftrightarrow in_2(e') \curvearrowleft_u in_1(e')$.

  J4 If $e \in E_{v \restriction Ck_3}$ and $e' \in E_{v \restriction Ck_6}$, then $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_u in_2(e') \Leftrightarrow in_2(e') \curvearrowleft_u in_1(e')$.

  J5 The only pointers in $\curvearrowleft_u$ are those specified by (J1)–(J4); that is,

  - if $[e \in E_{v \restriction Ck_1}$ and $e' \in E_{v \restriction Ck_4}]$ or $[e \in E_{v \restriction Ck_1}$ and $e' \in E_{v \restriction Ck_5}]$ or $[e \in E_{v \restriction Ck_1}$ and $e' \in E_{v \restriction Ck_6}]$ or $[e \in E_{v \restriction Ck_2}$ and $e' \in E_{v \restriction Ck_5}]$ or $[e \in E_{v \restriction Ck_2}$ and $e' \in E_{v \restriction Ck_6}]$ or $[e \in E_{v \restriction Ck_3}$ and $e' \in E_{v \restriction Ck_4}]$ or $[e \in E_{v \restriction Ck_4}$ and $e' \in E_{v \restriction Ck_4}]$ or $[e \in E_{v \restriction Ck_5}$ and $e' \in E_{v \restriction Ck_5}]$ or $[e \in E_{v \restriction Ck_6}$ and $e' \in E_{v \restriction Ck_6}]$, then $in_1(e) \not\curvearrowleft_u in_1(e')$ and $in_1(e) \not\curvearrowleft_u in_2(e')$
  - if $[e \in E_{v \restriction Ck_4}$ and $e' \in E_{v \restriction Ck_4}]$ or $[e \in E_{v \restriction Ck_5}$ and $e' \in E_{v \restriction Ck_5}]$ or $[e \in E_{v \restriction Ck_6}$ and $e' \in E_{v \restriction Ck_6}]$, then $in_2(e) \not\curvearrowleft_u in_2(e')$

  - if $[e \in E_{v\restriction Ck_4}$ and $e' \in E_{v\restriction Ck_1}]$ or $[e \in E_{v\restriction Ck_4}$ and $e' \in E_{v\restriction Ck_2}]$ or $[e \in E_{v\restriction Ck_4}$ and $e' \in E_{v\restriction Ck_3}]$ or $[e \in E_{v\restriction Ck_5}$ and $e' \in E_{v\restriction Ck_1}]$ or $[e \in E_{v\restriction Ck_5}$ and $e' \in E_{v\restriction Ck_2}]$ or $[e \in E_{v\restriction Ck_5}$ and $e' \in E_{v\restriction Ck_3}]$ or $[e \in E_{v\restriction Ck_6}$ and $e' \in E_{v\restriction Ck_1}]$ or $[e \in E_{v\restriction Ck_6}$ and $e' \in E_{v\restriction Ck_2}]$ or $[e \in E_{v\restriction Ck_6}$ and $e' \in E_{v\restriction Ck_3}]$, then $in_1(e) \not\curvearrowleft_u in_1(e')$ and $in_2(e) \not\curvearrowleft_u in_1(e')$
  - if $[e \in E_{v\restriction Ck_4}$ and $e' \in E_{v\restriction Ck_5}]$ or $[e \in E_{v\restriction Ck_4}$ and $e' \in E_{v\restriction Ck_6}]$ or $[e \in E_{v\restriction Ck_5}$ and $e' \in E_{v\restriction Ck_4}]$ or $[e \in E_{v\restriction Ck_5}$ and $e' \in E_{v\restriction Ck_6}]$ or $[e \in E_{v\restriction Ck_6}$ and $e' \in E_{v\restriction Ck_4}]$ or $[e \in E_{v\restriction Ck_6}$ and $e' \in E_{v\restriction Ck_5}]$ or $e \neq e' \in E_{v\restriction Ck_4}$ or $e \neq e' \in E_{v\restriction Ck_5}$ or $e \neq e' \in E_{v\restriction Ck_6}$, then $in_1(e) \not\curvearrowleft_u in_1(e')$ and $in_1(e) \not\curvearrowleft_u in_2(e')$ and $in_2(e) \not\curvearrowleft_u in_1(e')$ and $in_2(e) \not\curvearrowleft_u in_2(e')$.

We can show that $\preceq_u$ is a total preorder order and that u respects singularity using the same structure as the proof of Lemma 3.24.

Then, we have to show that u is justified. By inspection of the definition, we can see that the partial function $\curvearrowleft_u$ is total because every non-initial event has a pointer. In particular, for any $e' \in E_{u\restriction Ck_5}$, the existence of events e, e_1, e_2 in the definition is guaranteed since $v \restriction Ck_2 + Ck_1 \in id_{Ck}$ and $u \restriction Ck_3 + Ck_1 \in id_{Ck}$ and $u \restriction Ck_5 + Ck_2 \in id_{Ck}$.

Now, we show $(\forall e, e' \in E_u)(e \curvearrowleft_u e' \Rightarrow (e \preceq_u e$ and $\lambda_u(e) \vdash \lambda_u(e'))$. Suppose $e \curvearrowleft_u e'$. By definition of $\curvearrowleft_u$, we have the following cases.

- If $e, e' \in E_{u\restriction Ck_1+Ck_2+Ck_3}$, then $e \curvearrowleft_u e' \therefore in_1^{-1}(e) \curvearrowleft_s in_1^{-1}(e')$ by (J1) $\therefore in_1^{-1}(e) \preceq_s in_1^{-1}(e') \therefore e \preceq_u e'$ by (P1)
- If $e \in E_{u\restriction Ck_4'}$ and $e' \in E_{u\restriction Ck_4}$, then $e \curvearrowleft_u e' \therefore in_2^{-1}(e) = in_1^{-1}(e')$ by (J2) $\therefore e \preceq_u e'$ by (P4)
- If $e \in E_{u\restriction Ck_5'}$ and $e' \in E_{u\restriction Ck_5}$, then $e \curvearrowleft_u e' \therefore in_2^{-1}(e) = in_1^{-1}(e')$ by (J3) $\therefore e \preceq_u e'$ by (P4)
- If $e \in E_{u\restriction Ck_6'}$ and $e' \in E_{u\restriction Ck_6}$, then $e \curvearrowleft_u e' \therefore in_2^{-1}(e) = in_1^{-1}(e')$ by (J4) $\therefore e \preceq_u e'$ by (P4)
- If $e \in E_{u\restriction Ck_2}$ and $e' \in E_{u\restriction Ck_4'}$, then $e \curvearrowleft_u e' \therefore in_1^{-1}(e) \curvearrowleft_v in_2^{-1}(e')$ by (J2) $\therefore in_1^{-1}(e) \preceq_v in_2^{-1}(e') \therefore e \preceq_u e'$ by (P2)
- If $e \in E_{u\restriction Ck_3}$ and $e' \in E_{u\restriction Ck_5'}$, then $e \curvearrowleft_u e' \therefore (\exists e_1 \in E_{v\restriction Ck_1}, \exists e_2 \in E_{v\restriction Ck_2})(e_1 \curvearrowleft_v in_1^{-1}(e)$ and $e_1 \curvearrowleft_v e_2$ and $e_2 \curvearrowleft_v in_2^{-1}(e'))$ by (J3). Since Ck is a single-event signature,

in each trace $s \in id_{Ck}$, we have by the definition of id that $e \frown_s e'$ implies $e \approx_s e'$. As $v \restriction Ck_2 + Ck_1 \in id_{Ck}$ and $v \restriction Ck_3 + Ck_1 \in id_{Ck}$ and $u \restriction Ck_2 + Ck_5 \in id_{Ck}$, we get $in_1^{-1}(e) \approx_v e_1 \approx_v e_2 \approx_v in_2^{-1}(e')$ $\therefore e \preceq_u e'$ by (P2)

- If $e \in E_{u\restriction Ck_3}$ and $e' \in E_{u\restriction Ck'_6}$, then $e \frown_u e' \therefore in_1^{-1}(e) \frown_v in_2^{-1}(e')$ by (J4) $\therefore in_1^{-1}(e) \preceq_v in_2^{-1}(e')$ $\therefore e \preceq_u e'$ by (P2)

Note that in each case $\lambda_u(e) \vdash \lambda_u(e')$.

We need to verify that $u \in \alpha_{Ck,Ck,Ck} \lightning id_{Ck} \otimes sp_{Ck} \lightning sp_{Ck}$.

- $u \restriction Ck_2 + Ck_1 \in id_{Ck}$ and $u \restriction Ck_3 + Ck_1 \in id_{Ck}$ follow from (P1), (J1) and the fact that $v \restriction Ck_2 + Ck_1 \in id_{Ck}$ and $v \restriction Ck_3 + Ck_1 \in id_{Ck}$.
- $u \restriction Ck'_4 + Ck_2 \in id_{Ck}$ follows from (P2), (P3), (J2) and $v \restriction Ck_4 + Ck_2 \in id_{Ck}$.
- $u \restriction Ck'_6 + Ck_3 \in id_{Ck}$ follows from (P2), (P3), (J4) and $v \restriction Ck_6 + Ck_3 \in id_{Ck}$.
- $u \restriction Ck_4 + Ck'_4 \in id_{Ck}$ follows from (P4), (J2) and the fact that $\preceq_v$ is reflexive.
- $u \restriction Ck_5 + Ck'_5 \in id_{Ck}$ follows from (P4), (J3) and the fact that $\preceq_v$ is reflexive.
- $u \restriction Ck_6 + Ck'_6 \in id_{Ck}$ follows from (P4), (J4) and the fact that $\preceq_v$ is reflexive.

We still have to show $u \restriction Ck'_5 + Ck_3 \in id_{Ck}$. Let $e \in E_{u\restriction Ck'_5}$. It follows that there is $e_5 \in E_{v\restriction Ck_5}$ such that $e = in_2(e_5)$ and $in_1(e_5) \in u \restriction Ck_5$ and $e \approx_u in_1(e_5)$ and $e \frown_{u\restriction Ck_5+Ck'_5} in_1(e_5)$. Let us refer to the previous fact by $(\Diamond)$. For all $e_5 \in E_{v\restriction Ck_5}$, we have the following facts.

- Since $v \restriction Ck_5 + Ck_2 \in id_{Ck}$, there is $e_2 \in E_{v\restriction Ck_2}$ such that $e_5 \leftrightharpoons_{v\restriction Ck_5+Ck_2} e_2$.
- As $v \restriction Ck_2 + Ck_1 \in id_{Ck}$, there is $e_1 \in E_{v\restriction Ck_1}$ such that $e_2 \leftrightharpoons_{v\restriction Ck_2+Ck_1} e_1$.
- Because $v \restriction Ck_3 + Ck_1 \in id_{Ck}$, there is $e_3 \in E_{v\restriction Ck_1}$ such that $e_1 \leftrightharpoons_{v\restriction Ck_3+Ck_1} e_3$.

Since $\preceq_v$ is transitive, $e_2 \approx_v e_5$. We use (P2) and (P3) to find that $e_2 \approx_u in_1(e_5)$ and $e_2 \approx_u in_2(e_5)$. We use $(\Diamond)$ to find that for all $e \in E_{u\restriction Ck'_5}$ there is $e_2 \in E_{u\restriction Ck_2}$ such that $e \leftrightharpoons_{u\restriction Ck'_5+Ck_2} e_2$. We use the same reasoning to conclude that for all $e_2 \in E_{u\restriction Ck_2}$ there is $e \in E_{u\restriction Ck'_5}$ such that $e \leftrightharpoons_{u\restriction Ck'_5+Ck_2} e_2$. In essence, for all $e \in E_{u\restriction Ck'_5+Ck_2}$, there is $e' \in E_{u\restriction Ck'_5+Ck_2}$ such that $e \leftrightharpoons_{u\restriction Ck'_5+Ck_2} e'$.

We now show that $\curvearrowleft_{v\restriction Ck_5+Ck_3}$ conforms to Definition 5.11. This can be noted by inspecting the definition of $\curvearrowleft_u$.

We now need to show that $u \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1 = v \restriction Ck_4 + Ck_5 + Ck_6 + Ck_1$. From the definition of u and the definition of projection, we see that $E_{u\restriction Ck_4+Ck_5+Ck_6+Ck_1} = in_1(E_{v\restriction Ck_4+Ck_5+Ck_6+Ck_1})$ and $\lambda_{u\restriction Ck_4+Ck_5+Ck_6+Ck_1} = \lambda_{v\restriction Ck_4+Ck_5+Ck_6+Ck_1} \circ in_1$. We can also verify that $(\forall e, e' \in E_{v\restriction Ck_4+Ck_5+Ck_6+Ck_1})(e \preceq_{v\restriction Ck_4+Ck_5+Ck_6+Ck_1} e' \Leftrightarrow in_1(e) \preceq_{u\restriction Ck_4+Ck_5+Ck_6+Ck_1} in_1(e'))$. Next, we prove that for all $e, e' \in E_{v\restriction Ck_4+Ck_5+Ck_6+Ck_1}$, we have $e \curvearrowleft_{v\restriction Ck_4+Ck_5+Ck_6+Ck_1} e' \Leftrightarrow in_1(e) \curvearrowleft_{u\restriction Ck_4+Ck_5+Ck_6+Ck_1} in_1(e')$ through a case study on the labels of e, e'

- Case: $\lambda_u(e) \in L_{Ck_1}$ and $\lambda_u(e') \in L_{Ck_4}$. This is similar to the following case. We use (J1) and (J2).

- Case: $\lambda_u(e) \in L_{Ck_1}$ and $\lambda_u(e') \in L_{Ck_5}$. Let $e \curvearrowleft_{u\restriction Ck_4+Ck_5+Ck_6+Ck_1} e'$. So, there are $e_3 \in E_{u\restriction Ck_3}$ and $e_5 \in E_{u\restriction Ck_5'}$ such that $e \curvearrowleft_u e_3$ and $e_3 \curvearrowleft_u e_5$ and $e_5 \curvearrowleft_u e'$. Using (J3) on $e_3 \curvearrowleft_u e_5$, there are $e_1 \in E_{v\restriction Ck_1}, e_2 \in E_{v\restriction Ck_2}$ such that $e_1 \curvearrowleft_v in_1^{-1}(e_3)$ and $e_1 \curvearrowleft_v e_2$ and $e_2 \curvearrowleft_v in_2^{-1}(e_5)$. Using (J1) on $e \curvearrowleft_u e_3$, we get $in_1^{-1}(e) \curvearrowleft_v in_1^{-1}(e_3)$. Therefore, $e_1 = in_1^{-1}(e)$. Using (J3) on $e_5 \curvearrowleft_u e'$, we get $in_2^{-1}(e_5) = in_1^{-1}(e')$. So, $in_1^{-1}(e) \curvearrowleft_v e_2$ and $e_2 \curvearrowleft_v in_1^{-1}(e')$. So, $in_1^{-1}(e) \curvearrowleft_v in_1^{-1}(e')$. For the opposite direction, let $e \curvearrowleft_{v\restriction Ck_4+Ck_5+Ck_6+Ck_1} e'$. So, there is $e_2 \in E_{v\restriction Ck_2}$ such that $e \curvearrowleft_v e_2$ and $e_2 \curvearrowleft_v e'$. Since $v \restriction Ck_3 + Ck_1 \in id_{Ck}$, there is $e_3 \in E_{v\restriction Ck_3}$ such that $e \curvearrowleft_v e_3$. By (J1), $in_1(e) \curvearrowleft_u in_1(e_3)$. By (J3), $in_1(e_3) \curvearrowleft_u in_2(e')$ and $in_2(e') \curvearrowleft_u in_1(e')$. So, $in_1(e) \curvearrowleft_{v\restriction Ck_4+Ck_5+Ck_6+Ck_1} in_1(e')$.

- Case: $\lambda_u(e) \in L_{Ck_1}$ and $\lambda_u(e') \in L_{Ck_6}$. This is similar to the previous case. We use (J1) and (J4).

- All other label assignments are illegal.

The proofs of (6.2), (6.3) and (6.4) are omitted but follow the same structure as the proof of (6.1). ∎

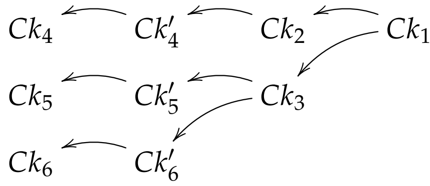


Figure 6.2: Justification pointers in traces $u \in \alpha_{Ck,Ck,Ck}; id_{Ck} \otimes sp_{Ck}; sp_{Ck}$ in Lemma 6.4

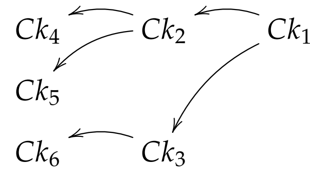


Figure 6.3: Justification pointers in traces $v \in sp_{Ck} \otimes id_{Ck}; sp_{Ck}$ in Lemma 6.4

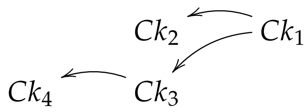


Figure 6.4: Justification pointers in traces $u \in cs_{Ck} \otimes id_{Ck}; sp_{Ck}$ in Lemma 6.4

Figure 6.5: Justification pointers in traces $v \in \lambda_{Ck}$ in Lemma 6.4

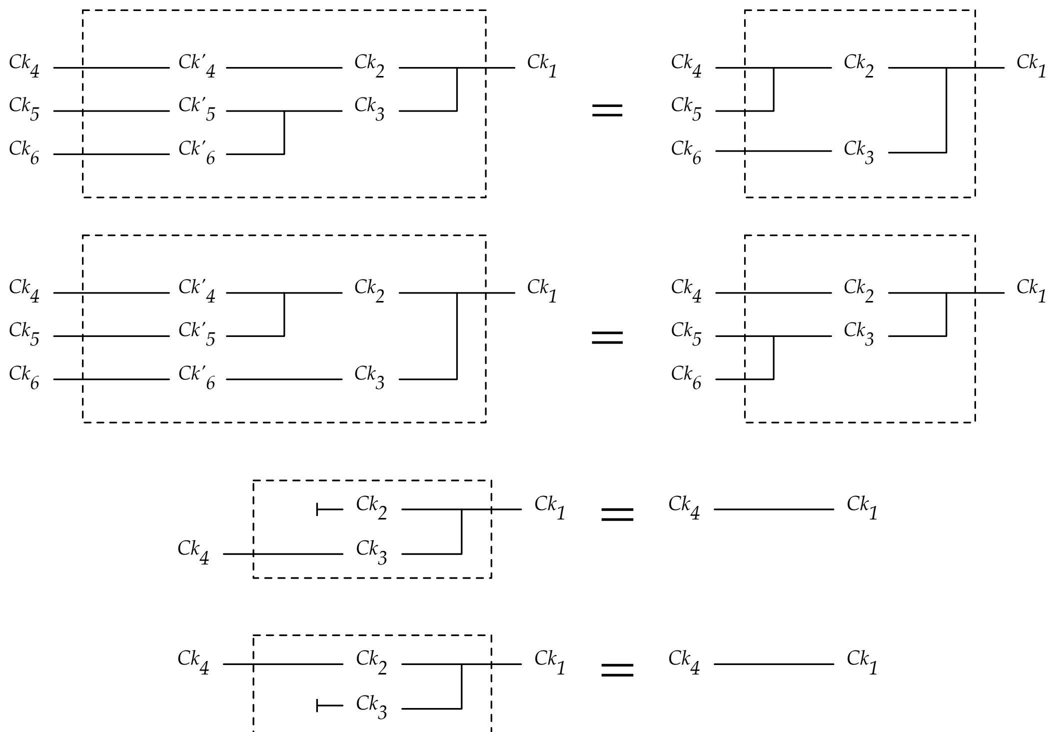


Figure 6.6: A diagrammatic representation of the monoid axioms

Lemma 6.5. *The triple $\langle Ck, sp_{Ck}, cs_{Ck}\rangle$ is a commutative monoid in* **SynProc**$_p$.

Proof. We need to prove that $\gamma_{Ck_1,Ck_2}; sp_{Ck} = sp_{Ck}$. We have

$$LHS = \{u \in \Theta_J(((Ck_2 \otimes Ck_3) \Rightarrow Ck_3' \otimes Ck_2') \Rightarrow Ck_1) \mid u \upharpoonright Ck_3' + Ck_1 \in id_{Ck} \text{ and}$$
$$u \upharpoonright Ck_2' + Ck_1 \in id_{Ck} \text{ and } u \upharpoonright Ck_2 + Ck_2' \in id_{Ck} \text{ and } u \upharpoonright Ck_3 + Ck_3' \in id_{Ck}\} \upharpoonright Ck_2 + Ck_3 + Ck_1$$
$$RHS = \{v \in \Theta_J(Ck_2 \otimes Ck_3 \Rightarrow Ck_1) \mid u \upharpoonright Ck_2 + Ck_1 \in id_{Ck} \text{ and } u \upharpoonright Ck_3 + Ck_1 \in id_{Ck}\}$$

We begin by proving that $\gamma_{Ck_1,Ck_2}; sp_{Ck} \subseteq sp_{Ck}$. Let $u \in \gamma_{Ck_1,Ck_2} ⨟ sp_{Ck}$. We find a trace $v \in sp_{Ck}$ such that $u \upharpoonright Ck_2 + Ck_3 + Ck_1 = v$. We define v as follows.

- $E_v = E_{u\upharpoonright Ck_2+Ck_3+Ck_1}$
- $\lambda_v = \lambda_{u\upharpoonright Ck_2+Ck_3+Ck_1}$
- $\preceq_v = \preceq_{u\upharpoonright Ck_2+Ck_3+Ck_1}$
- $\curvearrowleft_v$ is defined as follows, for all $e, e' \in E_u$.

  JV1 If $e \in E_{u\upharpoonright Ck_1}$ and $e' \in E_{u\upharpoonright Ck_2}$, then $(\exists e'' \in E_{u\upharpoonright Ck_2'})(e \curvearrowleft_u e''$ and $e'' \curvearrowleft_u e') \Leftrightarrow e \curvearrowleft_v e'$.

  JV2 If $e \in E_{u\upharpoonright Ck_1}$ and $e' \in E_{u\upharpoonright Ck_3}$, then $(\exists e'' \in E_{u\upharpoonright Ck_3'})(e \curvearrowleft_u e''$ and $e'' \curvearrowleft_u e') \Leftrightarrow e \curvearrowleft_v e'$.

  JV3 The only pointers in $\curvearrowleft_v$ are those specified by (JV1)–(JV2); that is, if $[e \in E_{u\upharpoonright Ck_2}$ and $e' \in E_{u\upharpoonright Ck_1}]$ or $[e \in E_{u\upharpoonright Ck_2}$ and $e' \in E_{u\upharpoonright Ck_3}]$ or $[e \in E_{u\upharpoonright Ck_3}$ and $e' \in E_{u\upharpoonright Ck_1}]$ or $[e \in E_{u\upharpoonright Ck_3}$ and $e' \in E_{u\upharpoonright Ck_2}]$, then $e \not\curvearrowleft_v e'$.

We first show that v is justified, i.e. that $\curvearrowleft_v$: $E_v \setminus \{e \in E_v \mid \lambda_v(e) \in I_{Ck_1}\} \to E_v$ is total and $(\forall e, e' \in E_v)(e \curvearrowleft_v e' \Rightarrow (e \preceq_v e$ and $\lambda_v(e) \vdash \lambda_v(e'))$. By inspection of the definition, we can see that the partial function $\curvearrowleft_v$ is total because every non-initial event has a pointer. In particular, for any $e' \in E_{u\upharpoonright Ck_2}$, the existence of events e'' in the definition is guaranteed since $u \upharpoonright Ck_2' + Ck_1 \in id_{Ck}$ and $u \upharpoonright Ck_2 + Ck_2' \in id_{Ck}$. The same is true for events $e' \in E_{u\upharpoonright Ck_2}$.

Now, we show $(\forall e, e' \in E_v)(e \curvearrowleft_v e' \Rightarrow (e \preceq_v e$ and $\lambda_v(e) \vdash \lambda_v(e'))$. Suppose $e \curvearrowleft_v e'$. By definition of $\curvearrowleft_v$, we have the following cases.

- If $e \in E_{u\upharpoonright Ck_1}$ and $e' \in E_{u\upharpoonright Ck_2}$, then $e \curvearrowleft_v e' \therefore (\exists e'' \in E_{u\upharpoonright Ck_2'})(e \curvearrowleft_u e''$ and $e'' \curvearrowleft_u e') \therefore (\exists e'' \in E_{u\upharpoonright Ck_2'})(e \preceq_u e''$ and $e'' \preceq_u e') \therefore e \preceq_u e' \therefore e \preceq_v e'$.

- If $e \in E_{u \restriction Ck_1}$ and $e' \in E_{u \restriction Ck_3}$, then $e \curvearrowleft_v e' \therefore (\exists e'' \in E_{u \restriction Ck'_3})(e \curvearrowleft_u e''$ and $e'' \curvearrowleft_u e') \therefore (\exists e'' \in E_{u \restriction Ck'_3})(e \preceq_u e''$ and $e'' \preceq_u e') \therefore e \preceq_u e' \therefore e \preceq_v e'$.

Note that in each case $\lambda_v(e) \vdash \lambda_v(e')$.

If follows from its definition that v satisfies $v \restriction Ck_2 + Ck_1 \in id_{Ck}$ and $v \restriction Ck_3 + Ck_1 \in id_{Ck}$. So, $v \in sp_{Ck}$.

We now need to show that $u \restriction Ck_2 + Ck_3 + Ck_1 = v$. From the definition of v and the definition of projection, $E_v = E_{u \restriction Ck_2+Ck_3+Ck_1}$, $\lambda_v = \lambda_{u \restriction Ck_2+Ck_3+Ck_1}$ and $\preceq_{v \restriction Ck_2+Ck_3+Ck_1} = \preceq_{u \restriction Ck_2+Ck_3+Ck_1}$. Next, we prove that for all $e, e' \in E_{u \restriction Ck_2+Ck_3+Ck_1}$, we have $e \curvearrowleft_{u \restriction Ck_2+Ck_3+Ck_1} e' \Leftrightarrow e \curvearrowleft_v e'$ through a case study on the labels of e, e'.

- Case: $\lambda_u(e) \in L_{Ck_1}$ and $\lambda_u(e') \in L_{Ck_2}$. Let $e \curvearrowleft_{u \restriction Ck_2+Ck_3+Ck_1} e'$. So, there is $e_2 \in E_{u \restriction Ck'_2}$ such that $e \curvearrowleft_u e_2$ and $e_2 \curvearrowleft_u e'$. Using (JV1) we get $e \curvearrowleft_v e'$. For the opposite direction, let $e \curvearrowleft_v e'$. So, there is $e_2 \in E_{u \restriction Ck'_2}$ such that $e \curvearrowleft_u e_2$ and $e_2 \curvearrowleft_u e'$. By definition of projection, $e \curvearrowleft_{u \restriction Ck_2+Ck_3+Ck_1} e'$.
- Case: $\lambda_u(e) \in L_{Ck_1}$ and $\lambda_u(e') \in L_{Ck_3}$. This is similar to the previous case. We use (JV2).
- All other label assignments are illegal.

Next, we show that $\gamma_{Ck_1,Ck_2} ; sp_{Ck} \supseteq sp_{Ck}$. Let $v \in sp_{Ck}$. We find a trace $u \in \gamma_{Ck_1,Ck_2} \between sp_{Ck}$ such that $v = u \restriction Ck_2 + Ck_3 + Ck_1$. We define u using the same algorithm described in the second half of the proof of equation (6.1) in Lemma 6.4.

- $E_u = E_v + E_{v \restriction Ck_2+Ck_3}$
- $\lambda_u = \lambda_v + \lambda_{v \restriction Ck_2+Ck_3}$
- $\preceq_u$ is defined as follows for all $e, e' \in E_v$.

  P1 If $e, e' \in E_v$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e')$

  P2 If $e \in E_v$ and $e' \in E_{v \restriction Ck_2+Ck_3}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_2(e')$

  P3 If $e \in E_{v \restriction Ck_2+Ck_3}$ and $e' \in E_v$, then $e \preceq_v e' \Leftrightarrow in_2(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_1(e')$

  P4 If $e, e' \in E_{v \restriction Ck_2+Ck_3}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_2(e) \preceq_u in_2(e') \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_2(e) \preceq_u in_1(e')$

- $\curvearrowleft_u$ is defined as follows, for all $e, e' \in E_v$.

  J1 If $e \in E_{v\upharpoonright Ck_1}$ and $e' \in E_{v\upharpoonright Ck_2}$, then $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_u in_2(e') \Leftrightarrow in_2(e') \curvearrowleft_u in_1(e')$.

  J2 If $e \in E_{v\upharpoonright Ck_1}$ and $e' \in E_{v\upharpoonright Ck_3}$, then $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_u in_2(e') \Leftrightarrow in_2(e') \curvearrowleft_u in_1(e')$.

  J3 The only pointers in $\curvearrowleft_v$ are those specified by (J1)–(J2); that is,

  - if $[e \in E_{v\upharpoonright Ck_2}$ and $e' \in E_{v\upharpoonright Ck_1}]$ or $[e \in E_{v\upharpoonright Ck_3}$ and $e' \in E_{v\upharpoonright Ck_1}]$, then $in_1(e) \not\curvearrowleft_v in_1(e')$ and $in_2(e) \not\curvearrowleft_v in_1(e')$.
  - if $e, e' \in E_{v\upharpoonright Ck_2}$ or $e, e' \in E_{v\upharpoonright Ck_3}$, then $in_1(e) \not\curvearrowleft in_1(e')$ and $in_1(e) \not\curvearrowleft in_2(e')$ and $in_2(e) \not\curvearrowleft in_2(e')$.
  - if $[e \in E_{v\upharpoonright Ck_2}$ and $e' \in E_{v\upharpoonright Ck_3}]$ or $[e \in E_{v\upharpoonright Ck_3}$ and $e' \in E_{v\upharpoonright Ck_2}]$ or $e \neq e' \in E_{v\upharpoonright Ck_2}$ or $e \neq e' \in E_{v\upharpoonright Ck_3}$, then $in_1(e) \not\curvearrowleft_v in_1(e')$ and $in_1(e) \not\curvearrowleft_v in_2(e')$ and $in_2(e) \not\curvearrowleft_v in_1(e')$ and $in_2(e) \not\curvearrowleft_v in_2(e')$.

We can show that $\preceq_u$ is a total preorder order and that u respects singularity using the same structure as the proof of Lemma 3.24. Then, we can show that u is justified following the same structure as the second half of the proof of equation (6.1) in Lemma 6.4.

We need to verify that $u \in \gamma_{Ck_1,Ck_2} \lightning sp_{Ck}$.

- $u \upharpoonright Ck_2 + Ck_2' \in id_{Ck}$ follows from (P4), (J1) and the fact that $\preceq_v$ is reflexive.
- $u \upharpoonright Ck_3 + Ck_3' \in id_{Ck}$ follows from (P4), (J2) and the fact that $\preceq_v$ is reflexive.
- $u \upharpoonright Ck_2' + Ck_1 \in id_{Ck}$ follows from (P1), (P4), (J1) and the fact that $v \upharpoonright Ck_2 + Ck_1 \in id_{Ck}$
- $u \upharpoonright Ck_3' + Ck_1 \in id_{Ck}$ follows from (P1), (P4), (J2) and the fact that $v \upharpoonright Ck_3 + Ck_1 \in id_{Ck}$

We now show that $\curvearrowleft_{v\upharpoonright Ck_5+Ck_3}$ conforms to Definition 5.11. This can be noted by inspecting the definition of $\curvearrowleft_u$.

Next, we need to show that $u \upharpoonright Ck_2 + Ck_3 + Ck_1 = v$. From the definition of u and the definition of projection, we see that $E_{u\upharpoonright Ck_2+Ck_3+Ck_1} = in_1(E_v)$ and $\lambda_{u\upharpoonright Ck_2+Ck_3+Ck_1} = \lambda_v \circ in_1$. We can also verify that $(\forall e, e' \in E_{v\upharpoonright Ck_2+Ck_3+Ck_1})(e \preceq_v e' \Leftrightarrow in_1(e) \preceq_{u\upharpoonright Ck_2+Ck_3+Ck_1} in_1(e'))$. Next, we prove that for all $e, e' \in E_{v\upharpoonright Ck_2+Ck_3+Ck_1}$, we have $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_{u\upharpoonright Ck_2+Ck_3+Ck_1} in_1(e')$ through a case study on the labels of e, e'

- Case: $\lambda_v(e) \in L_{Ck_1}$ and $\lambda_v(e') \in L_{Ck_2}$. Let $e \curvearrowleft_v e'$. By (J1), $in_1(e) \curvearrowleft_u in_2(e')$ and $in_2(e') \curvearrowleft_u in_1(e')$. By definition of projection, $e \curvearrowleft_{u\restriction Ck_2+Ck_3+Ck_1} e'$. For the opposite direction, let $e \curvearrowleft_{u\restriction Ck_2+Ck_3+Ck_1} e'$. So, there is $e_2 \in E_{u\restriction Ck'_2}$ such that $e \curvearrowleft_u e_2$ and $e_2 \curvearrowleft_u e'$. Using (J1) we get $e \curvearrowleft_v e'$.
- Case: $\lambda_v(e) \in L_{Ck_1}$ and $\lambda_v(e') \in L_{Ck_3}$. This is similar to the previous case. We use (J2).
- All other label assignments are illegal. ■

Before demonstrating that $\langle T, \eta, \mu\rangle$ is a monad, we state the following lemmas. The first one follows from the naturality of α.

Lemma 6.6. *Let $\sigma : A \to B$, $\tau : C \to D$, $\upsilon : E \to F$ be causal processes. We have $\sigma \otimes (\tau \otimes \upsilon) = \alpha^{-1}_{A,C,E}; (\sigma \otimes \tau) \otimes \upsilon; \alpha_{B,D,F}$ and $(\sigma \otimes \tau) \otimes \upsilon = \alpha_{A,C,E}; \sigma \otimes (\tau \otimes \upsilon); \alpha^{-1}_{B,D,F}$.*

The other two appear in the literature.

Lemma 6.7 ([JS93, Prop. 1.1])**.** $\rho_{A\otimes B} = \alpha_{A,B,I}; id_A \otimes \rho_B$.

Lemma 6.8 ([JS93, Prop. 1.1])**.** $\lambda_{A\otimes B} = \alpha^{-1}_{I,A,B}; \lambda_A \otimes id_B$.

Lemma 6.9. *$\langle T, \eta, \mu\rangle$ is a monad.*

Proof. We need to prove the following.

$$T(\mu_A); \mu_A = \mu_{TA}; \mu_A \tag{6.5}$$

$$T(\eta_A); \mu_A = id_{TA} \tag{6.6}$$

$$\eta_{TA}; \mu_A = id_{TA} \tag{6.7}$$

For (6.5), we have

$$\begin{aligned} T(\mu_A); \mu_A &= \alpha_{A,Ck,Ck} \otimes id_{Ck}; (id_A \otimes sp_{Ck}) \otimes id_{Ck}; \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck} \quad \text{by definition} \\ &= \alpha_{A,Ck,Ck} \otimes id_{Ck}; \alpha_{A,Ck\otimes Ck,Ck}; id_A \otimes (sp_{Ck} \otimes id_{Ck}); \alpha^{-1}_{A,Ck,Ck}; \alpha_{A,Ck,Ck}; \\ &\quad id_A \otimes sp_{Ck} \quad \text{by Lemma 6.6} \\ &= \alpha_{A,Ck,Ck} \otimes id_{Ck}; \alpha_{A,Ck\otimes Ck,Ck}; id_A \otimes (sp_{Ck} \otimes id_{Ck}); id_A \otimes sp_{Ck} \end{aligned}$$

$$= \alpha_{A,Ck,Ck} \otimes id_{Ck}; \alpha_{A,Ck\otimes Ck,Ck}; id_A \otimes (sp_{Ck} \otimes id_{Ck}; sp_{Ck}) \quad \text{by functoriality of } \otimes$$

$$= \alpha_{A,Ck,Ck} \otimes id_{Ck}; \alpha_{A,Ck\otimes Ck,Ck}; id_A \otimes (\alpha_{Ck,Ck,Ck}; id_{Ck} \otimes sp_{Ck}; sp_{Ck}) \quad \text{monoid laws}$$

$$= \alpha_{A,Ck,Ck} \otimes id_{Ck}; \alpha_{A,Ck\otimes Ck,Ck}; id_A \otimes \alpha_{Ck,Ck,Ck}; id_A \otimes (id_{Ck} \otimes sp_{Ck});$$
$$id_A \otimes sp_{Ck} \quad \text{by functoriality of } \otimes$$

$$= \alpha_{A\otimes Ck,Ck,Ck}; \alpha_{A,Ck,Ck\otimes Ck}; id_A \otimes (id_{Ck} \otimes sp_{Ck}); id_A \otimes sp_{Ck} \quad \text{by coherence of } \alpha$$

$$= \alpha_{A\otimes Ck,Ck,Ck}; \alpha_{A,Ck,Ck\otimes Ck}; \alpha^{-1}_{A,Ck,Ck\otimes Ck}; (id_A \otimes id_{Ck}) \otimes sp_{Ck}; \alpha_{A,Ck,Ck};$$
$$id_A \otimes sp_{Ck} \quad \text{by Lemma 6.6}$$

$$= \alpha_{A\otimes Ck,Ck,Ck}; (id_A \otimes id_{Ck}) \otimes sp_{Ck}; \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck}$$

$$= \mu_{TA}; \mu_A$$

For (6.6), we have

$$T(\eta_A); \mu_A = (\rho_A^{-1}; id_A \otimes cs_{Ck}) \otimes id_{Ck}; \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck} \quad \text{by definition}$$

$$= \rho_A^{-1} \otimes id_{Ck}; (id_A \otimes cs_{Ck}) \otimes id_{Ck}; \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck} \quad \text{by functoriality of } \otimes$$

$$= \rho_A^{-1} \otimes id_{Ck}; \alpha_{A,I,Ck}; id_A \otimes (cs_{Ck} \otimes id_{Ck}); \alpha^{-1}_{A,Ck,Ck}; \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck} \quad \text{by Lemma 6.6}$$

$$= \rho_A^{-1} \otimes id_{Ck}; \alpha_{A,I,Ck}; id_A \otimes (cs_{Ck} \otimes id_{Ck}); id_A \otimes sp_{Ck}$$

$$= \rho_A^{-1} \otimes id_{Ck}; \alpha_{A,I,Ck}; id_A \otimes (cs_{Ck} \otimes id_{Ck}; sp_{Ck}) \quad \text{by functoriality of } \otimes$$

$$= \rho_A^{-1} \otimes id_{Ck}; \alpha_{A,I,Ck}; id_A \otimes \lambda_{Ck} \quad \text{monoid laws}$$

$$= \rho_A^{-1} \otimes id_{Ck}; \rho_A \otimes id_{Ck} \quad \text{coherence of } \rho$$

$$= (\rho_A^{-1}; \rho_A) \otimes id_{Ck} \quad \text{by functoriality of } \otimes$$

$$= id_A \otimes id_{Ck}$$

$$= id_{A\otimes Ck} \quad \text{by functoriality of } \otimes$$

$$= id_{TA}$$

For (6.7), we have

$$\eta_{TA}; \mu_A = \rho^{-1}_{A\otimes Ck}; id_{A\otimes Ck} \otimes cs_{Ck}; \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck} \quad \text{by definition}$$

$$= \rho^{-1}_{A\otimes Ck}; (id_A \otimes id_{Ck}) \otimes cs_{Ck}; \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck} \quad \text{by functoriality of } \otimes$$

$$= \rho^{-1}_{A\otimes Ck}; \alpha_{A,Ck,I}; id_A \otimes (id_{Ck} \otimes cs_{Ck}); \alpha^{-1}_{A,Ck,Ck}; \alpha_{A,Ck,Ck}; id_A \otimes sp_{Ck} \quad \text{by Lemma 6.6}$$

$$= \rho^{-1}_{A\otimes Ck}; \alpha_{A,Ck,I}; id_A \otimes (id_{Ck} \otimes cs_{Ck}); id_A \otimes sp_{Ck}$$

$$= \rho^{-1}_{A\otimes Ck}; \alpha_{A,Ck,I}; id_A \otimes (id_{Ck} \otimes cs_{Ck}; sp_{Ck}) \quad \text{by functoriality of } \otimes$$

$$= \rho^{-1}_{A\otimes Ck}; \alpha_{A,Ck,I}; id_A \otimes \rho_{Ck} \quad \text{monoid laws}$$

$$= \rho^{-1}_{A\otimes Ck}; \rho_{A\otimes Ck} \quad \text{by Lemma 6.7}$$

$$= id_{A\otimes Ck}$$

$$= id_{TA}$$

■

We define the component of the strength of the clock monad as $t_{A,B} = \alpha^{-1}_{A,B,Ck}$. Its costrength natural transformation has canonical component $t'_{A,B} = \gamma_{TA,B}; t_{B,A}; T(\gamma_{B,A})$ [Jac94].

The following two lemmas are immediate from the coherence of the symmetry γ [JS93].

Lemma 6.10. $t'_{A,B} = \gamma_{A\otimes Ck,B}; \alpha^{-1}_{B,A,Ck}; \gamma_{B,A} \otimes id_{Ck} = \alpha_{A,Ck,B}; id_A \otimes \gamma_{Ck,B}; \alpha^{-1}_{A,B,Ck}$

Lemma 6.11. $\gamma_{A,B\otimes C} = \alpha^{-1}_{A,B,C}; \gamma_{A,B} \otimes id_C; \alpha_{B,A,C}; id_B \otimes \gamma_{A,C}; \alpha^{-1}_{B,C,A}$.

We can now show that the clock monad is a commutative strong monad.

Theorem 6.12. *$\langle T, \eta, \mu, t\rangle$ is a commutative strong monad.*

Proof. We need to prove the following.

$$t_{A\otimes B,C}; T(\alpha_{A,B,C}) = \alpha_{A,B,TC}; id_A \otimes t_{B,C}; t_{A,B\otimes C} \tag{6.8}$$

$$t_{I,A}; T(\lambda_A) = \lambda_{TA} \tag{6.9}$$

$$id_A \otimes \eta_B; t_{A,B} = \eta_{A\otimes B} \tag{6.10}$$

$$id_A \otimes \mu_B; t_{A,B} = t_{A,TB}; T(t_{A,B}); \mu_{A\otimes B} \tag{6.11}$$

$$t_{TA,B}; T(t'_{A,B}); \mu_{A\otimes B} = t'_{A,TB}; T(t_{A,B}); \mu_{A\otimes B} \tag{6.12}$$

For (6.8), we have

$$LHS = \alpha^{-1}_{A\otimes B,C,Ck}; \alpha_{A,B,C} \otimes id_{Ck}$$

$$= \alpha_{A,B,C\otimes Ck}; id_A \otimes \alpha^{-1}_{B,C,Ck}; \alpha^{-1}_{A,B\otimes C,Ck} \quad \text{by coherence of } \alpha$$

$$= RHS$$

For (6.9), we use Lemma 6.8. For (6.10), we have

$$\begin{aligned}
LHS &= id_A \otimes (\rho_B^{-1}; id_B \otimes cs_{Ck}); \alpha^{-1}_{A,B,Ck} \\
&= id_A \otimes \rho_B^{-1}; id_A \otimes (id_B \otimes cs_{Ck}); \alpha^{-1}_{A,B,Ck} \quad \text{by functoriality of } \otimes \\
&= id_A \otimes \rho_B^{-1}; \alpha^{-1}_{A,B,I}; (id_A \otimes id_B) \otimes cs_{Ck}; \alpha_{A,B,Ck}; \alpha^{-1}_{A,B,Ck} \quad \text{by Lemma 6.6} \\
&= id_A \otimes \rho_B^{-1}; \alpha^{-1}_{A,B,I}; id_{A\otimes B} \otimes cs_{Ck} \quad \text{by functoriality of } \otimes \\
&= \rho^{-1}_{A\otimes B}; id_{A\otimes B} \otimes cs_{Ck} \quad \text{by Lemma 6.7} \\
&= RHS
\end{aligned}$$

For (6.11), we have

$$\begin{aligned}
LHS &= id_A \otimes (\alpha_{B,Ck,Ck}; id_B \otimes sp_{Ck}); \alpha^{-1}_{A,B,Ck} \\
&= id_A \otimes \alpha_{B,Ck,Ck}; id_A \otimes (id_B \otimes sp_{Ck}); \alpha^{-1}_{A,B,Ck} \quad \text{by functoriality of } \otimes \\
&= id_A \otimes \alpha_{B,Ck,Ck}; \alpha^{-1}_{A,B,Ck\otimes Ck}; (id_A \otimes id_B) \otimes sp_{Ck}; \alpha_{A,B,Ck}; \alpha^{-1}_{A,B,Ck} \quad \text{by Lemma 6.6} \\
&= id_A \otimes \alpha_{B,Ck,Ck}; \alpha^{-1}_{A,B,Ck\otimes Ck}; id_{A\otimes B} \otimes sp_{Ck} \quad \text{by functoriality of } \otimes \\
&= \alpha^{-1}_{A,B\otimes Ck,Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck} \quad \text{by coherence of } \alpha \\
&= RHS
\end{aligned}$$

For (6.12), we have

$$\begin{aligned}
RHS &= \gamma_{A\otimes Ck,B\otimes Ck}; \alpha^{-1}_{B\otimes Ck,A,Ck}; \gamma_{B\otimes Ck,A} \otimes id_{Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck} \\
&= \alpha_{A,Ck,B\otimes Ck}; id_A \otimes \gamma_{Ck,B\otimes Ck}; \alpha^{-1}_{A,B\otimes Ck,Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck} \\
&\qquad \text{by Lemma 6.10} \\
&= \alpha_{A,Ck,B\otimes Ck}; id_A \otimes (\alpha^{-1}_{Ck,B,Ck}; \gamma_{Ck,B} \otimes id_{Ck}; \alpha_{B,Ck,Ck}; id_B \otimes \gamma_{Ck,Ck}; \alpha^{-1}_{B,Ck,Ck}); \alpha^{-1}_{A,B\otimes Ck,Ck}; \\
&\quad \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck} \quad \text{by Lemma 6.11} \\
&= \alpha_{A,Ck,B\otimes Ck}; id_A \otimes \alpha^{-1}_{Ck,B,Ck}; id_A \otimes (\gamma_{Ck,B} \otimes id_{Ck}); id_A \otimes \alpha_{B,Ck,Ck}; id_A \otimes (id_B \otimes \gamma_{Ck,Ck}); \\
&\quad id_A \otimes \alpha^{-1}_{B,Ck,Ck}; \alpha^{-1}_{A,B\otimes Ck,Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck} \quad \text{by functoriality of } \otimes \\
&= \alpha_{A,Ck,B\otimes Ck}; id_A \otimes \alpha^{-1}_{Ck,B,Ck}; \alpha^{-1}_{A,Ck\otimes B,Ck}; (id_A \otimes \gamma_{Ck,B}) \otimes id_{Ck}; \alpha_{A,B\otimes Ck,Ck}; id_A \otimes \alpha_{B,Ck,Ck}; \\
&\quad \alpha^{-1}_{A,B,Ck\otimes Ck}; (id_A \otimes id_B) \otimes \gamma_{Ck,Ck}; \alpha_{A,B,Ck\otimes Ck}; id_A \otimes \alpha^{-1}_{B,Ck,Ck}; \alpha^{-1}_{A,B\otimes Ck,Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck};
\end{aligned}$$

$$\alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck} \quad \text{by Lemma 6.6}$$

$$= \alpha^{-1}_{A\otimes Ck,B,Ck}; \alpha_{A,Ck,B} \otimes id_{Ck}; (id_A \otimes \gamma_{Ck,B}) \otimes id_{Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes \gamma_{Ck,Ck};$$
$$id_{(A\otimes B)\otimes(Ck\otimes Ck)}; id_{A\otimes B} \otimes sp_{Ck}$$

by functoriality of $\otimes$ and the following equalities stemming from the coherence of α,

- $\alpha_{A,Ck,B\otimes Ck}; id_A \otimes \alpha^{-1}_{Ck,B,Ck}; \alpha^{-1}_{A,Ck\otimes B,Ck} = \alpha^{-1}_{A\otimes Ck,B,Ck}; \alpha_{A,Ck,B} \otimes id_{Ck}$ and
- $\alpha_{A,B\otimes Ck,Ck}; id_A \otimes \alpha_{B,Ck,Ck}; \alpha^{-1}_{A,B,Ck\otimes Ck} = \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}$ and
- $\alpha_{A,B,Ck\otimes Ck}; id_A \otimes \alpha^{-1}_{B,Ck,Ck}; \alpha^{-1}_{A,B\otimes Ck,Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck} = id_{(A\otimes B)\otimes(Ck\otimes Ck)}$.

We have

$$LHS = \alpha^{-1}_{A\otimes Ck,B,Ck}; (\gamma_{A\otimes Ck,B}; \alpha^{-1}_{B,A,Ck}; \gamma_{B,A} \otimes id_{Ck}) \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck}$$
$$= \alpha^{-1}_{A\otimes Ck,B,Ck}; (\alpha_{A,Ck,B}; id_A \otimes \gamma_{Ck,B}; \alpha^{-1}_{A,B,Ck}) \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck} \quad \text{by Lemma 6.10}$$
$$= \alpha^{-1}_{A\otimes Ck,B,Ck}; \alpha_{A,Ck,B} \otimes id_{Ck}; (id_A \otimes \gamma_{Ck,B}) \otimes id_{Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes sp_{Ck}$$
$$\text{by functoriality of } \otimes$$
$$= \alpha^{-1}_{A\otimes Ck,B,Ck}; \alpha_{A,Ck,B} \otimes id_{Ck}; (id_A \otimes \gamma_{Ck,B}) \otimes id_{Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck};$$
$$id_{A\otimes B} \otimes (\gamma_{Ck,Ck}; sp_{Ck}) \quad \text{by commutativity of the monoid}$$
$$= \alpha^{-1}_{A\otimes Ck,B,Ck}; \alpha_{A,Ck,B} \otimes id_{Ck}; (id_A \otimes \gamma_{Ck,B}) \otimes id_{Ck}; \alpha^{-1}_{A,B,Ck} \otimes id_{Ck}; \alpha_{A\otimes B,Ck,Ck}; id_{A\otimes B} \otimes \gamma_{Ck,Ck};$$
$$id_{A\otimes B} \otimes sp_{Ck} \quad \text{by functoriality of } \otimes$$
$$= RHS$$

■

The commutative strong monad $\langle T, \eta, \mu, t\rangle$ induces a symmetric monoidal monad [Koc72] $\langle T, \eta, \mu, m\rangle$, where m is a natural transformation, assigning to each pair of objects, A and B, a morphism $m_{A,B} : TA \otimes TB \to T(A \otimes B)$ defined as follows.

$$m_{A,B} = t_{TA,B}; T(t'_{A,B}); \mu_{A\otimes B} \cong id_A \otimes id_B \otimes sp_{Ck}$$

$$TA \otimes TB \xrightarrow{t_{TA,B}} T(TA \otimes B) \xrightarrow{Tt'_{A,B}} T(T(A \otimes B)) \xrightarrow{\mu_{A\otimes B}} T(A \otimes B)$$

Jacobs [Jac94] calls the natural transformation m a *double strength map*. Its existence corresponds to driving two processes from the same clock source in a unique way. This is illustrated in Figure 6.7.

Using the clock monad we can canonically derive a *Kleisli category* of processes equipped with a clock. Using standard definitions [Mac98], the Kleisli category $\mathbf{SynProc}_T$ is defined as follows.

- The objects of $\mathbf{SynProc}_T$ are those of $\mathbf{SynProc}_P$.
- The set of morphisms $\hom_{\mathbf{SynProc}_T}(A, B)$ is $\hom_{\mathbf{SynProc}_P}(A, TB)$.
- The composition of $\sigma : A \to TB$ and $\tau : B \to TC$ is defined by,

$$\sigma ;_T \tau = \sigma; T(\tau); \mu_C$$

- The identity morphism over A is η_A.

Diagrammatically, the Kleisli composition of processes $f : A \to TB$ and $g : B \to TC$ is depicted in Figure 6.8.

Lemma 6.13 ([Jac94, Theorem 4.3]). *If T is a commutative monad on a symmetric monoidal category $\mathbf{C}$, then the Kleisli category $\mathbf{C}_T$ is also a symmetric monoidal category, defined by*

- $A \otimes_T B = A \otimes B$
- $f \otimes_T g = (f \otimes g); m$
- $I_T = I \otimes Ck$

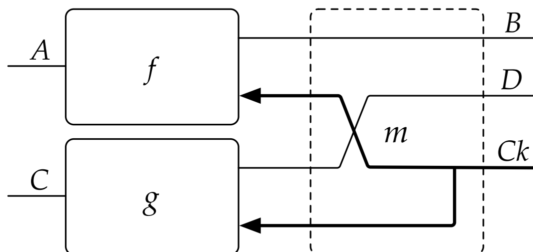


Figure 6.7: A circuit representation of the double strength map.

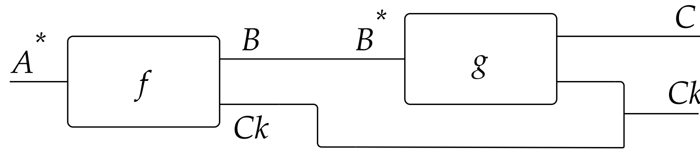


Figure 6.8: A circuit representation of composition in the Kleisli category $\mathbf{SynProc}_T$.

Corollary 6.14. $\mathbf{SynProc}_T$ *is a symmetric monoidal category.*

We will identify a subcategory $\mathbf{SynProc}'$ of $\mathbf{SynProc}_T$ wherein for each morphism $\sigma : A \to B$, initial events in A are always caused by B-events rather than Ck-events. We will call processes that satisfy this property *pure*.

Definition 6.15 (Pure Process)**.** *A process $\sigma : A \to B$ in $\mathbf{SynProc}_T$ is called* pure *when all $s \in \sigma$ satisfy the following. For all $e \in E_{s\restriction A}$, if $e' \curvearrowleft_s e$ and $\lambda_s(e) \in I_A$, then $\lambda_s(e') \in I_B$.*

This technical condition allows us to identify a closed monoidal subcategory of $\mathbf{SynProc}_T$. Incidentally, disallowing clock events from causing other events is akin to disallowing clock gating in digital design—a technique that allows controlling local clock signals using combinational logic [ZRM90].

Note that the identity in $\mathbf{SynProc}_T$ is pure by definition.

Lemma 6.16. *If $\sigma : A \to B$ and $\tau : B \to C$ in $\mathbf{SynProc}_T$ are pure, then $\sigma ;_T \tau$ is pure.*

Proof. Let $v \in \sigma;_T \tau$. It follows by definition of composition that there is a trace $u \in \sigma \between \tau \otimes id_{Ck} \between \mu_C$ over $\Theta_J(((A \Rightarrow B \otimes Ck_4) \Rightarrow (C \otimes Ck_2) \otimes Ck_3) \Rightarrow C' \otimes Ck_1)$ such that $u \restriction A + C' +$

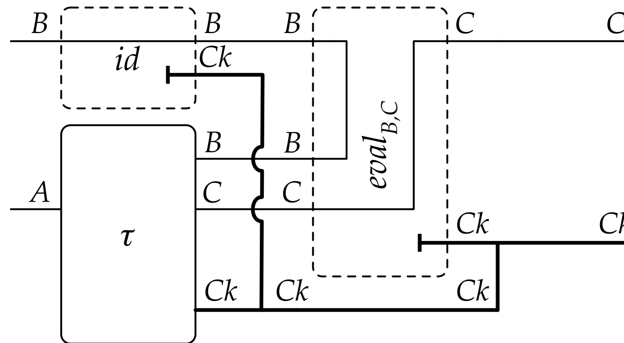


Figure 6.9: A circuit representation of $(id_B \otimes_T \tau) ;_T eval_{B,C\,T}$.

$Ck_1 = v$. Suppose $e' \curvearrowleft_v e$ and $\lambda_v(e) \in I_A$. By definition of composition, there are events $e_1 \in E_{u\upharpoonright C+Ck_2+Ck_3}, e_2 \in E_{u\upharpoonright B+Ck_4}$ such that $\lambda_u(e_1) \in I_{C\otimes Ck_2 \otimes Ck_3}$ and $\lambda_u(e_2) \in I_{B\otimes Ck_4}$ and $e' \curvearrowleft_u e_1$ and $e_1 \curvearrowleft_u e_2$ and $e_2 \curvearrowleft_u e$. We use the fact that σ is pure with $e_2 \curvearrowleft_u e$ and $\lambda_u(e) \in I_A$ to conclude that $\lambda_u(e_2) \in I_B$. Then, from $e_1 \curvearrowleft_u e_2$ and $\lambda_u(e_2) \in I_B$ we find that $\lambda_u(e_1) \in I_C$ since τ is pure. Finally, we use the definition of identity (Definition 5.11) on $e' \curvearrowleft_u e_1$ and $\lambda_u(e_1) \in I_C$ to conclude that $\lambda_u(e') \in I_{C'}$. ■

Lemma 6.17. *If $\sigma : A \to B$ and $\tau : C \to D$ in* $\mathbf{SynProc}_T$ *are pure, then $\sigma \otimes_T \tau$ is pure.*

Proof. We use the same strategy used in the proof of Lemma 6.16. ■

The various monoidal isomorphisms are given by tensoring (in the sense of Definition 3.31) $\alpha_{A,B,C}, \lambda_A, \rho_A, \gamma_{A,B}$ and $eval_{A,B}$ in $\mathbf{SynProc}_P$ with cs_{Ck}. These are pure as tensoring does not alter justification pointers.

Proposition 6.18. $\mathbf{SynProc}'$ *is a symmetric monoidal closed category.*

The proof follows the same structure as Proposition 3.42. A circuit representation of the universal property is depicted in Figure 6.9.

Note that the monad construction is flexible and general enough to allow the definition of different clock domains using different clocks and clock monads. In the following section, we will identify a category of clocked processes. Later, we study determinism in the context of clocked processes.

6.2 Clocked Processes

In the preceding section, we saw that the clock monad does not force processes to use the clock. However, this means that processes may contain traces where a clock event does not occur in every round; for example, $\langle e_1, tick\rangle \cdot e_2 \cdot \langle e_2, tick\rangle$. Such traces have no obvious interpretation. In a globally synchronous setting, the clock is universal and therefore, all events fall within its domain. Moreover, the clock divides time into logical segments with no interceding gaps. So, it is not possible for events to occur between two consecutive clock cycles.

We will therefore work with the collection of processes in **SynProc**$'$ that synchronise with the global clock Ck. That is, those that can only receive inputs and produce outputs simultaneously with the clock reserved event *tick*. In other words, we want to remove traces that do not correspond to a physical realisation of the clock: those where the clock event does not occur in every round. We therefore define a category **CkProc** of clocked processes. Its objects are those of **SynProc**$'$. We begin by defining *clocked traces*. These are justified traces where each event is simultaneous with a clock event.

Definition 6.19 (Clocked trace). *A* clocked trace *s over A is a justified trace satisfying for all $e \in E_s$, there exists $e' \in E_s$ such that $e \approx_s e'$ and $\lambda_s(e') = tick$.*

We denote by $\Theta_C(A)$ the set of clocked traces over A. In the following proofs, we sometimes include the clock object in the signature for extra clarity.

Definition 6.20 (Clocked process). *A* clocked process *$\sigma : A \to B$ is a causal pure process only containing clocked traces over $\Theta_J(A \Rightarrow B \otimes Ck)$.*

Note that all clocked processes are also processes in **SynProc**$'$.

For any signatures A, B and C we define the set of *interaction traces*, written $int_C(A, B, C)$ as $\Theta_C((A \Rightarrow B) \Rightarrow C)$.

Definition 6.21 (Interaction of clocked processes). *The interaction $\sigma \between_C \tau$ of clocked processes $\sigma : A \to B$ and $\tau : B \to C$ is the set $\{u \in int_C(A, B, C) \mid u \restriction A + B + Ck \in \sigma$ and $u \restriction B + C + Ck \in \tau\}$.*

Definition 6.22 (Composition of clocked processes). *Let $\sigma : A \to B$ and $\tau : B \to C$ be clocked processes. Their composition $\sigma;_C \tau$ is the set $\{u \restriction A + C + Ck \mid u \in \sigma \between_C \tau\}$.*

Intuitively, the clocked identity is the largest clocked process contained in η_A, where η_A is the identity process in **SynProc**$_T$. It is defined as follows.

Definition 6.23 (Clocked identity). *The identity morphism for object A, denoted by id_A, is the clocked process consisting of traces u over $A_1 \Rightarrow A_2$ satisfying, for all events e in $E_u \setminus \{e \in E_u \mid \lambda_u(e) \in lin(Ck)\}$, there exists an event $e' \neq e$, such that*

1. *$e \approx_u e'$ and*

2. $[\lambda_u(e) = inl(a)$ *if and only if* $\lambda_u(e') = inr(a)]$ *and* $[\lambda_u(e) = inr(a)$ *if and only if* $\lambda_u(e') = inl(a)]$ *and*

3. (a) *if* $\lambda_u(e) \in I_{A_1}$, *then* $e' \curvearrowleft_u e$

   (b) *if* $\lambda_u(e) \in I_{A_2}$, *then* $e \curvearrowleft_u e'$

   (c) *if* $\lambda_u(e) \notin I_{A_1}$ *and* $\lambda(e) \notin I_{A_2}$, *then there are* $e_1, e_2 \in E_u$ *such that* $e_1 \curvearrowleft_u e$ *and* $e_2 \curvearrowleft_u e'$ *and* $e_1 \approx_u e_2$ *and* $[\lambda_u(e_1) = inl(a')$ *if and only if* $\lambda_u(e_2) = inr(a')]$ *and* $[\lambda_u(e_1) = inr(a')$ *if and only if* $\lambda_u(e_2) = inl(a')]$

where $a, a' \in L_A$. *For any two* $e, e' \in E_u$, *if conditions 1–3 hold, we write* $e \rightleftarrows_u e'$.

On objects, the tensor product is defined exactly as in **SynProc**$'$. On morphisms, it is defined as follows.

Definition 6.24 (Tensor product). *The tensor product* $\sigma \otimes_C \tau$ *of clocked processes* $\sigma : A \to B$ *and* $\tau : C \to D$ *is the set* $\{u \in \Theta_C(A \otimes C \Rightarrow B \otimes D) \mid u \restriction A + B + Ck \in \sigma$ *and* $u \restriction C + D + Ck \in \tau\}$.

Lemma 6.25. *For clocked processes* $\sigma : A \to B$ *and* $\tau : B \to C$, *we have* $\sigma \fatsemi_C \tau = (\sigma \fatsemi T(\tau) \fatsemi \mu_C) \restriction A + B + C + Ck$.

Proof. Showing that $\sigma \fatsemi_C \tau \supseteq (\sigma \fatsemi T(\tau) \fatsemi \mu_C) \restriction A + B + C + Ck$ is straightforward. So, we only prove that $\sigma \fatsemi_C \tau \subseteq (\sigma \fatsemi T(\tau) \fatsemi \mu_C) \restriction A + B + C + Ck$.

$$LHS = \{v \in \Theta_C((A \Rightarrow B) \Rightarrow C \otimes Ck) \mid u \restriction A + B + Ck \in \sigma \text{ and } u \restriction B + C + Ck \in \tau\}$$

$$\begin{aligned} RHS = \{u \in \Theta_C(((A \Rightarrow B \otimes Ck_4) \Rightarrow (C' \otimes Ck_2) \otimes Ck_3) \Rightarrow C \otimes Ck) \mid u \restriction A + B + Ck_4 \in \sigma \\ \text{and } u \restriction B + C' + Ck_2 \in \tau \text{ and } u \restriction Ck_4 + Ck_3 \in id_{Ck} \text{ and } u \restriction C' + C \in id_C \\ \text{and } u \restriction Ck_2 + Ck \in id_{Ck} \text{ and } u \restriction Ck_3 + Ck \in id_{Ck}\} \end{aligned}$$

Take $v \in \sigma \fatsemi_C \tau$. We construct a trace $u \in \sigma \fatsemi T(\tau) \fatsemi \mu_C$ such that $u \restriction A + B + C + Ck = v$ as follows.

- $E_u = E_v + E_{v \restriction Ck} + E_{v \restriction Ck} + E_{v \restriction C + Ck}$.
- $\lambda_u = \lambda_v + \lambda_{v \restriction Ck} + \lambda_{v \restriction Ck} + \lambda_{v \restriction C + Ck}$.

- The preorder $\preceq_u$ is the relation satisfying the following, for all $e, e' \in E_v$.

  P1 if $e, e' \in E_{v\restriction A+B}$, then $in_1(e) \preceq_u in_1(e')$

  P2 if $e \in E_{v\restriction A+B}$ and $e' \in E_{v\restriction Ck}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_1(e) \preceq_u in_3(e') \Leftrightarrow in_1(e) \preceq_u in_4(e')$

  P3 if $e \in E_{v\restriction A+B}$ and $e' \in E_{v\restriction C}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_4(e')$

  P4 if $e \in E_{v\restriction Ck}$ and $e \in E_{v\restriction A+B}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_2(e) \preceq_u in_1(e') \Leftrightarrow in_3(e) \preceq_u in_1(e') \Leftrightarrow in_4(e) \preceq_u in_1(e')$

  P5 if $e \in E_{v\restriction C}$ and $e \in E_{v\restriction A+B}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_4(e) \preceq_u in_1(e')$

  P6 if $e \in E_{v\restriction Ck}$ and $e \in E_{v\restriction C}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_4(e') \Leftrightarrow in_2(e) \preceq_u in_1(e') \Leftrightarrow in_2(e) \preceq_u in_4(e') \Leftrightarrow in_3(e) \preceq_u in_1(e') \Leftrightarrow in_3(e) \preceq_u in_4(e') \Leftrightarrow in_4(e) \preceq_u in_1(e') \Leftrightarrow in_4(e) \preceq_u in_4(e')$

  P7 if $e \in E_{v\restriction C}$ and $e \in E_{v\restriction Ck}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_1(e) \preceq_u in_3(e') \Leftrightarrow in_1(e) \preceq_u in_4(e') \Leftrightarrow in_4(e) \preceq_u in_1(e') \Leftrightarrow in_4(e) \preceq_u in_2(e') \Leftrightarrow in_4(e) \preceq_u in_3(e') \Leftrightarrow in_4(e) \preceq_u in_4(e')$

  P8 if $e, e' \in E_{v\restriction C}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_4(e') \Leftrightarrow in_4(e) \preceq_u in_1(e') \Leftrightarrow in_4(e) \preceq_u in_4(e')$

  P9 if $e, e' \in E_{v\restriction Ck}$, then $e \preceq_v e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_1(e) \preceq_u in_3(e') \Leftrightarrow in_1(e) \preceq_u in_4(e') \Leftrightarrow in_2(e) \preceq_u in_1(e') \Leftrightarrow in_2(e) \preceq_u in_2(e') \Leftrightarrow in_2(e) \preceq_u in_3(e') \Leftrightarrow in_2(e) \preceq_u in_4(e') \Leftrightarrow in_3(e) \preceq_u in_1(e') \Leftrightarrow in_3(e) \preceq_u in_2(e') \Leftrightarrow in_3(e) \preceq_u in_3(e') \Leftrightarrow in_3(e) \preceq_u in_4(e') \Leftrightarrow in_4(e) \preceq_u in_1(e') \Leftrightarrow in_4(e) \preceq_u in_2(e') \Leftrightarrow in_4(e) \preceq_u in_3(e') \Leftrightarrow in_4(e) \preceq_u in_4(e')$

- Let $\curvearrowleft_u$ be defined as follows, for all $e, e' \in E_v$.

  J1 If $e, e' \in E_{v\restriction A+B}$, then $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_u in_1(e')$.

  J2 If $e, e' \in E_{v\restriction C}$, then $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_u in_1(e') \Leftrightarrow in_4(e) \curvearrowleft_u in_4(e')$, and $\lambda_v(e) \in I_C \Leftrightarrow in_4(e) \curvearrowleft_u in_1(e)$.

  J3 If $e \in E_{v\restriction C}$ and $e' \in E_{v\restriction B}$, then $e \curvearrowleft_v e' \Leftrightarrow in_1(e) \curvearrowleft_u in_4(e) \Leftrightarrow in_4(e) \curvearrowleft_u in_1(e')$.

  J4 If $e \in E_{v\restriction Ck}$, then $in_1(e) \curvearrowleft_u in_2(e) \Leftrightarrow in_1(e) \curvearrowleft_u in_3(e) \Leftrightarrow in_3(e) \curvearrowleft_u in_4(e)$.

  J5 The only pointers in $\curvearrowleft_u$ are those specified by (J1)–(J4); that is,

- if $e \in E_{v\restriction A}$ and $e' \in E_{v\restriction B}$, then $in_1(e) \not\frown_u in_1(e')$,
- if $e \in E_{v\restriction A+B}$ and $e' \in E_{v\restriction C}$, then $in_1(e) \not\frown_u in_1(e')$ and $in_1(e) \not\frown_u in_4(e')$,
- if $e \in E_{v\restriction A+B}$ and $e' \in E_{v\restriction Ck}$, then $in_1(e) \not\frown_u in_1(e')$ and $in_1(e) \not\frown_u in_2(e')$ and $in_1(e) \not\frown_u in_3(e')$ and $in_1(e) \not\frown_u in_4(e')$,
- if $e \in E_{v\restriction C}$ and $e' \in E_{v\restriction A}$, then $in_1(e) \not\frown_u in_1(e')$ and $in_4(e) \not\frown_u in_1(e')$,
- if $e \in E_{v\restriction C}$ and $e' \in E_{v\restriction B}$, then $in_1(e) \not\frown_u in_1(e')$,
- if $e \in E_{v\restriction C}$ and $e' \in E_{v\restriction Ck}$, then $in_1(e) \not\frown_u in_1(e')$ and $in_1(e) \not\frown_u in_2(e')$ and $in_1(e) \not\frown_u in_3(e')$ and $in_1(e) \not\frown_u in_4(e')$ and $in_4(e) \not\frown_u in_1(e')$ and $in_4(e) \not\frown_u in_2(e')$ and $in_4(e) \not\frown_u in_3(e')$ and $in_4(e) \not\frown_u in_4(e')$,
- if $e, e' \in E_{v\restriction Ck}$, then $in_1(e) \not\frown_u in_1(e')$ and $in_1(e) \not\frown_u in_4(e')$ and $in_2(e) \not\frown_u in_1(e')$ and $in_2(e) \not\frown_u in_2(e')$ and $in_2(e) \not\frown_u in_3(e')$ and $in_2(e) \not\frown_u in_4(e')$ and $in_3(e) \not\frown_u in_1(e')$ and $in_3(e) \not\frown_u in_2(e')$ and $in_3(e) \not\frown_u in_3(e')$ and $in_4(e) \not\frown_u in_1(e')$ and $in_4(e) \not\frown_u in_2(e')$ and $in_4(e) \not\frown_u in_3(e')$ and $in_4(e) \not\frown_u in_4(e')$,
- if $e \in E_{v\restriction Ck}$ and $e' \in E_{v\restriction A+B}$, then $in_1(e) \not\frown_u in_1(e')$ and $in_2(e) \not\frown_u in_1(e')$ and $in_3(e) \not\frown_u in_1(e')$ and $in_4(e) \not\frown_u in_1(e')$,
- if $e \in E_{v\restriction Ck}$ and $e' \in E_{v\restriction C}$, then $in_1(e) \not\frown_u in_1(e')$ and $in_2(e) \not\frown_u in_1(e')$ and $in_3(e) \not\frown_u in_1(e')$ and $in_4(e) \not\frown_u in_1(e')$ and $in_1(e) \not\frown_u in_4(e')$ and $in_2(e) \not\frown_u in_4(e')$ and $in_3(e) \not\frown_u in_4(e')$ and $in_4(e) \not\frown_u in_4(e')$.

We first need to verify that $\preceq_u$ is a total preorder, that u respects singularity and that u is justified. These proofs follow the same structure as e.g. the second half of the proof of equation (6.1) in Lemma 6.4.

We additionally need to check that u is clocked. Take $e \in E_u$.

- $\lambda_u(e) \in L_A$ or $\lambda_u(e) \in L_B$ or $\lambda_u(e) \in L_C$. So, $e = in_1(e')$ and $e' \approx_v c$ where c is a clock event. We use (P2) and (P3) to find that $e \approx_u in_1(c)$.
- $\lambda_u(e) \in L_{C'}$. So, $e = in_4(e')$ and $e' \approx_v c$ where c is a clock event. We use (P6) and (P7) to find that $e \approx_u in_4(c)$.
- All cases where e is a clock event are trivial because $\preceq_u$ is reflexive.

Next we need to verify that u satisfies the conditions set earlier.

- $u \restriction C' + C \in id_C$ follows from (P8), (J2) and the fact that $\preceq_v$ is reflexive.
- $u \restriction Ck_4 + Ck_3 \in id_{Ck}$ follows from (P9), (J4) and the fact that $\preceq_v$ is reflexive.
- $u \restriction Ck_2 + Ck \in id_{Ck}$ follows from (P9), (J4) and the fact that $\preceq_v$ is reflexive.
- $u \restriction Ck_3 + Ck \in id_{Ck}$ follows from (P9), (J4) and the fact that $\preceq_v$ is reflexive.
- $u \restriction A + B + Ck_4 \in \sigma$. We define a bijection $\phi : E_{v\restriction A+B+Ck} \to E_{u\restriction A+B+Ck_4}$ as follows.

$$\phi(e) = \begin{cases} in_1(e) & \text{if } e \in E_{v\restriction A+B} \\ in_4(e) & \text{if } e \in E_{v\restriction Ck} \end{cases}$$

  The proof that the conditions in Definition 5.4 are met follows from the definitions of E_u and λ_u; (P1); (P2); (P4); (P9); (J1) and (J4).

- $u \restriction B + C' + Ck_2 \in \tau$ We define a bijection $\phi : E_{v\restriction B+C+Ck} \to E_{u\restriction B+C'+Ck_2}$ as follows.

$$\phi(e) = \begin{cases} in_1(e) & \text{if } e \in E_{v\restriction B} \\ in_4(e) & \text{if } e \in E_{v\restriction C} \\ in_2(e) & \text{if } e \in E_{v\restriction Ck_2} \end{cases}$$

  The proof that the conditions in Definition 5.4 are met follows from the definitions of E_u and λ_u; (P3); (P4); (P5); (P6); (P7); (P9); (J1) and (J4).

Finally, we prove that $u \restriction A + B + C + Ck = v$. From the definition of u and the definition of projection, we see that $E_{u\restriction A+B+C+Ck} = in_1(E_v)$ and $\lambda_{u\restriction A+B+C+Ck} = \lambda_v \circ in_1$. We can also verify, by inspection of the definition of $\preceq_u$ that $(\forall e, e' \in E_v)(e \preceq_v e' \Leftrightarrow in_1(e) \preceq_{u\restriction A+B+C+Ck} in_1(e'))$. Next, we prove that for all $e, e' \in E_v$, we have $e \frown_v e' \Leftrightarrow in_1(e) \frown_{u\restriction A+B+C+Ck} in_1(e')$ through a case study on the labels of $e, e' \in E_v$.

- Case: $\lambda_v(e) \in L_C$ and $\lambda_v(e') \in L_C$. By (J2), $in_1(e) \frown_u in_1(e')$ so $in_1(e) \frown_{u\restriction A+B+C+Ck} in_1(e')$.
- Case: $\lambda_v(e) \in L_B$ and $\lambda_v(e') \in L_B$. By (J1), $in_1(e) \frown_u in_1(e')$ so $in_1(e) \frown_{u\restriction A+B+C+Ck} in_1(e')$.

- Case: $\lambda_v(e) \in L_A$ and $\lambda_v(e') \in L_A$. By (J1), $in_1(e) \curvearrowleft_u in_1(e')$ so $in_1(e) \curvearrowleft_{u \upharpoonright A+B+C+Ck} in_1(e')$.

- Case: $\lambda_v(e) \in L_C$ and $\lambda_v(e') \in L_B$. By (J3), $in_1(e) \curvearrowleft_u in_4(e)$ and $in_4(e) \curvearrowleft_u in_1(e')$ so $in_1(e) \curvearrowleft_{u \upharpoonright A+B+C+Ck} in_1(e')$.

- Case: $\lambda_v(e) \in L_B$ and $\lambda_v(e') \in L_A$. By (J1), $in_1(e) \curvearrowleft_u in_1(e')$ so $in_1(e) \curvearrowleft_{u \upharpoonright A+B+C+Ck} in_1(e')$.

- All other label assignments are illegal. ■

Corollary 6.26. *For clocked processes $\sigma : A \to B$ and $\tau : B \to C$, we have $\sigma;_C \tau = \sigma;_T \tau$.*

Following the same strategy as the previous lemma, we can prove the following.

Lemma 6.27. *For clocked processes $\sigma : A \to B$ and $\tau : C \to D$, we have $\sigma \otimes_C \tau = \sigma \otimes_T \tau$.*

The proofs of the following two lemmas follow straightforwardly from the definitions.

Lemma 6.28. *If $\sigma : A \to B$ and $\tau : B \to C$ are clocked processes, then $\sigma;_C \tau$ is a clocked process.*

Lemma 6.29. *If $\sigma : A \to B$ and $\tau : C \to D$ are clocked processes, then $\sigma \otimes_C \tau$ is a clocked process.*

We now show that the clocked identity is well defined.

Lemma 6.30. *The clocked identity is the identity for clocked processes*

Proof. Let $\sigma : A \to B$ be a clocked process. We have to prove that $\sigma;_C id_B = \sigma$. Since $id_B \subseteq \mu_B$ and since all clocked processes are also processes in $\mathbf{SynProc}_T$, it is easy to see that $\sigma;_C id_B \subseteq \sigma$. We then prove that $\sigma;_C id_B \supseteq \sigma$. The proof follows the same structure as the proof of Lemma 3.24.

Let $s \in \sigma$. We will find an interaction trace $u \in int_C(A, B, B')$ such that $u \upharpoonright A + B + Ck = s$ and $u \upharpoonright B + B' + Ck \in id_B$ and $u \upharpoonright A + B' + Ck = s$. Let u be defined as follows.

- $E_u = E_s + E_{s \upharpoonright B}$.

- $\lambda_u = \lambda_s + \lambda_{s \upharpoonright B}$.

- The preorder $\preceq_u$ is defined as follows. For all $e, e' \in E_s$,

P1 if $e, e' \in E_s$, then $e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_1(e')$

P2 if $e \in E_s$ and $e' \in E_{s\restriction B}$, then $e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_2(e')$

P3 if $e \in E_{s\restriction B}$ and $e' \in E_s$, then $e \preceq_s e' \Leftrightarrow in_2(e) \preceq_u in_1(e')$

P4 if $e, e' \in E_{s\restriction B}$, then $[e \preceq_s e' \Leftrightarrow in_1(e) \preceq_u in_1(e') \Leftrightarrow in_2(e) \preceq_u in_2(e') \Leftrightarrow in_1(e) \preceq_u in_2(e') \Leftrightarrow in_2(e) \preceq_u in_1(e')]$

- Define $\curvearrowleft_u$ as follows, for all $e, e' \in E_s$.

J1 If $e, e' \in E_{s\restriction A}$, then $e \curvearrowleft_s e' \Leftrightarrow in_1(e) \curvearrowleft_u in_1(e')$

J2 If $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $e \curvearrowleft_s e' \Leftrightarrow in_1(e) \curvearrowleft_u in_1(e')$

J3 If $e, e' \in E_{s\restriction B}$, then $e \curvearrowleft_s e' \Leftrightarrow in_1(e) \curvearrowleft_u in_1(e') \Leftrightarrow in_2(e) \curvearrowleft_u in_2(e')$

J4 $\lambda_s(e) \in I_B \Leftrightarrow in_2(e) \curvearrowleft_u in_1(e)$

J5 The only pointers in $\curvearrowleft_u$ are those specified by (J1)–(J4); that is,

- if $e \in E_{s\restriction A}$ and $e' \in E_{s\restriction B}$, then $[in_1(e) \not\curvearrowleft_u in_1(e')$ and $in_1(e) \not\curvearrowleft_u in_2(e')]$,
- if $e \in E_{s\restriction B}$, then $in_1(e) \not\curvearrowleft_u in_2(e')$,
- if $e \in E_{s\restriction B}$ and $e' \in E_{s\restriction A}$, then $in_2(e) \not\curvearrowleft_u in_1(e')$.

The proofs that $\preceq_u$ is a total preorder and u respects singularity follow the same structure as the proof of Lemma 3.24. We prove that u is justified. It is clear from (J1) and (J2) that all non-initials in u have a pointer. We verify that $(\forall e, e' \in E_u)(e \curvearrowleft_u e' \Rightarrow e \preceq_u e')$ through a case study. Suppose $e \curvearrowleft_u e'$. There are three possibilities given by the definition of $\curvearrowleft_u$.

- $in_1^{-1}(e) \curvearrowleft_s in_1^{-1}(e') \therefore in_1^{-1}(e) \preceq_s in_1^{-1}(e')$ because s is justified $\therefore e \preceq_u e'$ by (P1).
- $in_2^{-1}(e) \curvearrowleft_s in_2^{-1}(e') \therefore in_2^{-1}(e) \preceq_s in_2^{-1}(e')$ because s is justified $\therefore e \preceq_u e'$ by (P4).
- $in_2^{-1}(e) = in_1^{-1}(e')$ and $in_2^{-1}(e), in_1^{-1}(e') \in E_{s\restriction B} \therefore in_2^{-1}(e) \preceq_s in_1^{-1}(e')$ as $\preceq_s$ is reflexive $\therefore e \preceq_u e'$ by (P4).

The trace u satisfies purity by construction. We check that u is clocked. Take $e \in E_u$

- $\lambda_u(e) \in L_A$ or $\lambda_u(e) \in L_B$. So, $e = in_1(e')$ and $e' \approx_s c$ where c is a clock event. We use (P1) to find that $e \approx_u in_1(c)$.

- $\lambda_u(e) \in L_{B'}$. So, $e = in_2(e')$ and $e' \approx_s c$ where c is a clock event. We use (P2) and (P3) to find that $e \approx_u in_1(c)$.
- All cases where e is a clock event are trivial because $\preceq_u$ is reflexive.

Finally, we show that u satisfies the conditions set out above. First, note that since u is clocked, then $u \restriction A + B + Ck$ and $u \restriction B + B' + Ck$ and $u \restriction A + B' + Ck$ are clocked.

Because $\preceq_s$ is reflexive, (P4) implies that $(\forall e \in E_{s\restriction B})(in_1(e) \approx_u in_2(e))$. We also have $\lambda_{u\restriction B+B'}(in_1(e)) = inl(b)$ iff $\lambda_{u\restriction B+B'}(in_2(e)) = inr(b), b \in L_B$. Moreover, (J3) and (J4) ensure that pointers are assigned according to Definition 6.23. So, $u \restriction B + B' + Ck \in id_B$. We briefly argue that $u \restriction A + B = s$. We have $E_{u\restriction A+B+Ck} = in_1(E_s)$ and $\lambda_{u\restriction A+B+Ck} = \lambda_s \circ in_1$. From the definition of $\preceq_u$, we can see that $(\forall e, e' \in E_s)(e \preceq_s e' \Leftrightarrow in_1(e) \preceq_{u\restriction A+B+Ck} in_1(e')$. We need to prove that $(\forall e, e' \in E_s)(e \curvearrowleft_s e' \Leftrightarrow in_1(e) \curvearrowleft_{u\restriction A+B+Ck} in_1(e')$. This is clear from the definition of $\curvearrowleft_u$.

We then show that $u \restriction A + B' + Ck = s$. We have $E_{u\restriction A+B'+Ck} = E_{s\restriction A+Ck} + E_{s\restriction B}$. There is a bijection $\phi : E_s \to E_{u\restriction A+B+Ck}$ defined as

$$\phi(e) = \begin{cases} in_1(e) & \text{if } e \in E_{s\restriction A+Ck} \\ in_2(e) & \text{if } e \in E_{s\restriction B} \end{cases}$$

It follows from the definition of u that $\lambda_{s\restriction A+Ck} = \lambda_s \circ \phi$. The proofs that $(\forall e, e' \in E_s)(e \preceq_s e' \Leftrightarrow \phi(e) \preceq_{u\restriction A+B+Ck} \phi(e')$ and $(\forall e, e' \in E_s)(e \curvearrowleft_s e' \Leftrightarrow \phi(e) \preceq_{u\restriction A+B+Ck} \phi(e')$ are by case study on the labels of e, e' and follow straightforwardly from the definition of u.

The proof that $u \restriction A + B' = s$ is similar. ■

Next, we demonstrate that the tensor product preserves the identity.

Lemma 6.31. *The tensor product preserves the clocked identity.*

Proof. We want to show that $id_A \otimes_C id_B = id_{A\otimes B}$.

First, we prove $id_A \otimes_C id_B \subseteq id_{A\otimes B}$. Let $u \in id_A \otimes_C id_B$; that is, $u \in \Theta_C(A \otimes B \Rightarrow A' \otimes B' \otimes Ck)$ and $u \restriction A + A' + Ck \in id_A$ and $u \restriction B + B' + Ck \in id_B$. Expanding this using Definition 6.23, we get

- $u \in \Theta_C(A \otimes B \Rightarrow A' \otimes B \otimes Ck)$ and
- $(\forall e \in E_{u\restriction A+A'+Ck}, \exists e' \in E_{u\restriction A+A'+Ck})(e \rightleftarrows_{u\restriction A+A'+Ck} e')$ and
- $(\forall e \in E_{u\restriction A+A'+Ck}, \exists e' \in E_{u\restriction A+A'+Ck})(e \approx_{u\restriction A+A'+Ck} e'$ and $\lambda_{u\restriction A+A'+Ck}(e') = tick)$ and
- $(\forall e \in E_{u\restriction B+B'+Ck}, \exists e' \in E_{u\restriction B+B'+Ck})(e \rightleftarrows_{u\restriction B+B'+Ck} e')$.
- $(\forall e \in E_{u\restriction B+B'+Ck}, \exists e' \in E_{u\restriction B+B'+Ck})(e \approx_{u\restriction B+B'+Ck} e'$ and $\lambda_{u\restriction B+B'+Ck}(e') = tick)$

Since $E_u = E_{u\restriction A+A'+Ck} \cup E_{u\restriction B+B'+Ck}$ and $(\forall e, e' \in E_u)($if $e \approx_{u\restriction A+A'+Ck} e'$ or $e \approx_{u\restriction B+B'+Ck} e'$, then $e \approx_{u'} e')$ and for all $a \in E_{u\restriction A+A'+Ck}$, $b \in E_{u\restriction B+B'+Ck}$, we have $\lambda_{u\restriction A+A'+Ck}(a) = \lambda_u(a)$ and $\lambda_{u\restriction B+B'+Ck}(b) = \lambda_u(b)$, we conclude that for all $e \in E_u$, there is $e' \in E_u$ such that $e \rightleftarrows_u e'$ and for all $e \in E_u$ there is $e' \in E_u$ such that $e \approx_u e'$ and $\lambda_u(e') = tick$. Therefore, $u \in id_{A\otimes B}$.

Now, we prove $id_A \otimes id_B \supseteq id_{A\otimes B}$. Let $u \in id_{A\otimes B}$; that is, $u \in \Theta_C(A \otimes B \Rightarrow A' \otimes B' \otimes Ck)$ and $(\forall e \in E_u, \exists e \in E_u)(e \rightleftarrows_u e')$ and $(\forall e \in E_u, \exists e' \in E_u)(e \approx_u e'$ and $\lambda_u(e') = tick)$. After projecting u over $A + A' + Ck$, we get

- $u \restriction A + A' + Ck \in \Theta_C(A \Rightarrow A' \otimes Ck)$ and
- $(\forall e \in E_{u\restriction A+A'+Ck}, \exists e' \in E_{u\restriction A+A'+Ck})(e \rightleftarrows_{u\restriction A+A'} e')$ and
- $(\forall e \in E_{u\restriction A+A'+Ck}, \exists e' \in E_{u\restriction A+A'+Ck})(e \approx_{u\restriction A+A'+Ck} e'$ and $\lambda_{u\restriction A+A'+Ck}(e') = tick)$

that is, $u \restriction A + A' + Ck \in id_A$. Projecting u over $B + B' + Ck$ yields

- $u \restriction B + B' + Ck \in \Theta_C(B \Rightarrow B' \otimes Ck)$ and
- $(\forall e \in E_{u\restriction B+B'+Ck}, \exists e' \in E_{u\restriction B+B'+Ck})(e \rightleftarrows_{u\restriction B+B'+Ck} e')$ and
- $(\forall e \in E_{u\restriction B+B'+Ck}, \exists e' \in E_{u\restriction B+B'+Ck})(e \approx_{u\restriction B+B'+Ck} e'$ and $\lambda_{u\restriction B+B'+Ck}(e') = tick)$

that is, $u \restriction B + B' \in id_B$. So, $u \in id_A \otimes id_B$. ■

Lemma 6.32. **CkProc** *is a symmetric monoidal category.*

Proof. In addition to Lemmas 6.30 and 6.31, we need to verify the following.

- Associativity of composition. Take clocked processes $\sigma : A \to B$, $\tau : B \to C$ and $\gamma : C \to D$. We know that all clocked processes are processes in **SynProc**$'$, so $(\sigma;_T \tau);_T \gamma = \sigma;_T (\tau;_T \gamma)$. By Corollary 6.26, we get $(\sigma;_C \tau);_C \gamma = \sigma;_C (\tau;_C \gamma)$.

- Tensor preserves composition. Take clocked processes $\sigma : A \to B$, $\tau : B \to C$, $\alpha : D \to E$ and $\beta : E \to F$. We know that all clocked processes are processes in **SynProc**$'$, so $(\sigma;_T \tau) \otimes_T (\alpha;_T \beta) = (\sigma \otimes_T \alpha);_T (\tau \otimes_T \beta)$. By Corollary 6.26 and Lemma 6.27, we get $(\sigma;_C \tau) \otimes_C (\alpha;_C \beta) = (\sigma \otimes_C \alpha);_C (\tau \otimes_C \beta)$.

- The isomorphisms $\alpha_{A,B,C}, \lambda_A, \rho_A, \gamma_{A,B}$ are built using the clocked identity. The detailed argument follows the same structure as similar proofs in Section 3.2.1 ∎

We can now state the following.

Theorem 6.33. **CkProc** *is a symmetric monoidal closed category.*

The morphism $eval_{A,B}$ is built using the clocked identity. Then the proof, which is omitted, follows the structure of Proposition 3.42.

6.3 Observable Determinism

Building on the results of the previous section, we define and study the compositionality of deterministic processes.

Define $\mathit{final}(s)$, the set of final round events in a trace s, as follows.

$$\mathit{final}(s) = \{e \in E_s \mid (\forall e' \in E_s)(e' \preceq_s e)\}$$

We define $\mathit{final}^i(s), \mathit{final}^o(s)$ as the obvious restrictions to inputs and outputs, respectively. Let $rest(s)$ denote the trace s less the last round. Formally, $rest(s)$ is defined by the following.

- $E_{rest(s)} = E_s \setminus \mathit{final}(s)$

- $\lambda_{rest(s)} = \lambda_s \upharpoonright E_{rest(s)}$

- $\preceq_{rest(s)} = \preceq_s \cap E^2_{rest(s)}$

- $\curvearrowleft_{rest(s)} = \curvearrowleft_s \cap E^2_{rest(s)}$

In light of the standard definition of determinism [HO00], it is tempting to think of a deterministic synchronous process as one where all traces that are equal up to the penultimate round and receive the same inputs in the last round, must produce the same outputs. Intuitively,

$$\text{if } rest(s) = rest(s') \text{ and } \mathit{final}^i(s) = \mathit{final}^i(s'), \text{ then } \mathit{final}^o(s) = \mathit{final}^o(s') \tag{6.13}$$

However, such definition yields an erroneous characterisation of determinism. Consider the following example.

Example 6.34. Let processes $\sigma : A \to B$ and $\tau : B \to C$, with $L_A = \{a_1, a_2\}$, $L_B = \{b_1, b_2, b_3, b_4\}$, $\vdash_B = \{(b_1, b_2), (b_3, b_4)\}$, $L_C = \{c\}$, defined as follows.

$$\sigma = pc(\{ \langle\ b_1^i \overset{\curvearrowleft}{,} b_2^o\ ,\ a_1^o\ \rangle, \langle\ b_3^i \overset{\curvearrowleft}{,} b_4^o\ ,\ a_2^o\ \rangle \}$$

$$\tau = pc(\{\ c^i \overset{\curvearrowleft}{\cdot} \langle\ b_1^o \overset{\curvearrowleft}{,} b_2^i\ \rangle,\ c^i \overset{\curvearrowleft}{\cdot} \langle\ b_3^o \overset{\curvearrowleft}{,} b_4^i\ \rangle \}$$

According to (6.13), σ and τ are deterministic. However, their composition $\{c^i \cdot a_1^o, c^i \cdot a_2^o\}$ is not. The problem is that τ is erroneously identified as deterministic. Since both b_1 and b_3 are caused by c, their occurrence should be ruled as nondeterministic.

Our definition of deterministic process extends the usual definition (see for example [HO00]) to synchronous processes.

Definition 6.35 (Observably deterministic process)**.** *A process* $\sigma : A$ *is* observably deterministic *if in every distinct history, the output is uniquely determined with respect to the temporal order. That is, for all nonempty traces* $s, s' \in \sigma$ *such that* $rest(s) = rest(s')$*, where* $\phi : E_s \to E_{s'}$ *is the trace equivalence bijection, we have for every output event* o *in* $\mathit{final}(s)$ *and all inputs* $i \in E_s$ *and* $i' \in E_{s'}$*,*

1. *if* $i \curvearrowleft_s o$ *and* $i \not\approx_s o$*, then* $(\exists o' \in \mathit{final}(s'))(\lambda_s(o) = \lambda_{s'}(o')$ *and* $\phi(i) \curvearrowleft_{s'} o')$
2. *if* $i \curvearrowleft_s o$ *and* $i \approx_s o$ *and* $i' \in \mathit{final}(s')$ *and* $\lambda_s(i) = \lambda_{s'}(i')$*, then* $(\exists o' \in \mathit{final}(s'))(\lambda_s(o) = \lambda_{s'}(o')$ *and* $i' \curvearrowleft_{s'} o')$

Note that this definition only accounts for observable determinism; that is, it only uses information that can be observed about the process: the temporal order and justification pointers. This is not always an accurate characterisation of determinism as we discuss in the next section. In the remainder of the current section, whenever we mention determinism, we refer to the above definition.

In any trace of a clocked process, every event is simultaneous with the clock event *tick*. We can therefore use it as a counter. In each trace v, let $tick^v_n$ represent the nth clock event, where $n \in \mathbb{N}$, $[tick^v_n]$ the nth round, and v_n, the shortest prefix of v containing $tick^v_n$. By convention, we let $v_0 = \epsilon$.

Lemma 6.36. *For traces $s \in \sigma : A \to B$ and $t \in \tau : B \to C$, where σ and τ are clocked processes, there is at most one interaction $u \in \sigma \lightning \tau$ such that $u \upharpoonright A + B + Ck = s$ and $u \upharpoonright B + C + Ck = t$.*

Proof. By Definition 6.22, s and t synchronise on B-events and Ck-events. The are two cases.

- Either $s \upharpoonright B + Ck \neq t \upharpoonright B + Ck$ and therefore, $s \lightning t = \emptyset$
- Or, $s \upharpoonright B + Ck = t \upharpoonright B + Ck$; that is, s and t have the same number of rounds and they are driven by the same clock. In any interaction u of s and t, since $[tick^s_n] \upharpoonright B + Ck = [tick^t_n] \upharpoonright B + Ck$, we have $[tick^u_n] = ([tick^s_n] \upharpoonright A + B + Ck) + ([tick^t_n] \upharpoonright C)$.

It follows that when s and t interact, they produce a single trace. ■

For traces $s \in \sigma : A \to B$ and $\tau : B \to C$, where σ and τ are clocked processes, if $s \lightning t \neq \emptyset$, we say that s and t are *composable*. By slight abuse of notation, we refer to the unique trace in $s \lightning t$ by $s \lightning t$ and its projection over $A + C + Ck$ by $s;t$.

We now show that determinism is a compositional property in **CkProc**. In the following lemma, we show that the traces in the composition of two deterministic processes have unique origins in the components. This may seem surprising, but recall that all events have to synchronise with the global clock. In a locally synchronous setting, this lemma is obviously not correct.

Lemma 6.37. *Let $\sigma : A \to B$ and $\tau : B \to C$ be deterministic processes in* **CkProc**, *with traces s, s' in σ and t, t', in τ. If s and t are composable and $s;t = s';t'$, then $s = s'$ and $t = t'$.*

Proof. By Definition 6.22, we have that $s;t = s';t'$ implies,

$$(s \lightning t) \upharpoonright A + C + Ck = (s' \lightning t') \upharpoonright A + C + Ck \tag{6.14}$$

We prove that $s \lightning t = s' \lightning t'$ by induction over the length of traces s, t, s', t' given by the number of rounds n. Let $u = s \lightning t$ and $u' = s' \lightning t'$

- Base case $(n = 1)$. Suppose b is a B-event in $\mathit{final}(u)$. Let b be justified by an event e.
  - If b is initial in B then e is an input in $\mathit{final}(t)$. By (6.14), $e \in \mathit{final}(t')$. Since τ is deterministic, $b \in \mathit{final}(t')$.
  - If b is not initial in B, then because $\vdash_{A \Rightarrow B}$ and $\vdash_{B \Rightarrow C}$ are acyclic by Definition 5.1, there is a finite set of B-events $Z = \{e_1, \ldots, e_k, b\}$ such that $e_1 \curvearrowleft_u \ldots \curvearrowleft_u e_k \curvearrowleft_u b$ and e_1 is an initial B-event. We prove by induction that all events in Z are in $\mathit{final}(u')$.
    * Base case $(k = 1)$. We have $e_1^i \curvearrowleft_u b$. Then, there is a C-event $c \in \mathit{final}(t)$ such that $c \curvearrowleft_t e_1^o$. By (6.14), $c \in \mathit{final}(t')$. Since τ is deterministic, $e_1^o \in \mathit{final}(t')$. Since s' and t' compose, $e_1^i \in \mathit{final}(s')$. Since σ is deterministic, $b \in \mathit{final}(s')$.
    * Inductive step. Suppose all events $e_i \in Z$ where $i \in \{1, \ldots, m\}$ are in $\mathit{final}(u')$. Now we consider the next event e_{m+1} in Z. Suppose $e_{m+1} \in \mathit{final}(u)$ and $e_m \curvearrowleft_s e_{m+1}$. Let e_{m+1} be an output in $\mathit{final}(s)$. By the inductive hypothesis, $e_m \in \mathit{final}(s')$. Since σ is deterministic, we get $e_{m+1} \in \mathit{final}(s')$. Let e_{m+1} be an input in $\mathit{final}(s)$. Then, since s and t are composable, e_{m+1} is an output in $\mathit{final}(t)$. By the inductive hypothesis, $e_m \in \mathit{final}(t')$. Since τ is deterministic, we get $e_{m+1} \in \mathit{final}(t')$.
- Inductive step. Suppose $(s \lightning t)_n = (s' \lightning t')_n$. We show that rounds $n + 1$ are also equal in $s \lightning t$ and $s' \lightning t'$. Suppose b is a B-event in $\mathit{final}(u)$. Let b be justified by an event e. If $e \approx_u b$, then we use the same reasoning as the base case. In particular, e is either initial or is transitively justified by an initial event e_1 in the same round or in $\mathit{rest}(u)$. The base case already covers the first two cases, so it is sufficient to consider the case where $e \not\approx_u b$. If b is an output in s then $b \in \mathit{final}(s')$ because σ is deterministic and if b is an output in t then $b \in \mathit{final}(t')$ because τ is deterministic.

We need to verify that $(\forall e, e' \in E_u)(e \curvearrowleft_u e' \Leftrightarrow e \curvearrowleft_{u'} e')$.

- If e is a C-event and e' is a C-event, then $e \curvearrowleft_{u'} e'$ from (6.14).
- If e is a A-event and e' is a A-even, then $e \curvearrowleft_{u'} e'$ from (6.14).
- If e is a B-event and e' is a B-event, then if $e \approx_u e'$ we use the inductive argument above, otherwise $e \curvearrowleft_{u'} e'$ because σ and τ are deterministic.
- If e is a C-event and e' is a B-event, then e' is an output in t. We have $e \curvearrowleft_{u'} e$ since τ is deterministic.
- If e is a B-event and e' is a A-event, then e' is an output in s. We have $e \curvearrowleft_{u'} e$ since σ is deterministic. ■

We can now show that deterministic processes compose.

Lemma 6.38 (Compositionality). *If $\sigma : A \to B$ and $\tau : B \to C$ are deterministic processes in* **CkProc**, *then $\sigma;\tau$ is a deterministic process.*

Proof. Let $\sigma : A \to B$ and $\tau : B \to C$ be deterministic processes in **CkProc**. By Definition 6.35, for all $s, s' \in \sigma$ and $t, t' \in \tau$ such that $rest(s) = rest(s')$ and $rest(t) = rest(t')$ we have,

$$\text{if } i \curvearrowleft_s o \text{ and } i \not\approx_s o, \text{ then } o \in final(s') \tag{6.15}$$

$$\text{if } i \curvearrowleft_s o \text{ and } i \approx_s o \text{ and } i \in final(s'), \text{ then } o \in final(s') \tag{6.16}$$

$$\text{if } i \curvearrowleft_t o \text{ and } i \not\approx_t o, \text{ then } o \in final(t') \tag{6.17}$$

$$\text{if } i \curvearrowleft_t o \text{ and } i \approx_t o \text{ and } i \in final(t'), \text{ then } o \in final(t') \tag{6.18}$$

We show that $\sigma;\tau$ conforms to Definition 6.35.

Take traces v, v' in $\sigma;\tau$ such that $s;t = v$ and $s';t' = v'$ with $s, s' \in \sigma$ and $t, t' \in \tau$.

For the first part of the proof, we assume that $rest(v) = rest(v')$ and $i \curvearrowleft_v o$ and $i \not\approx_v o$ and show that $o \in final(v')$. Using Lemma 6.37, it follows that $rest(s) = rest(s')$ and $rest(t) = rest(t')$. From $i \curvearrowleft_v o$ and $i \not\approx_v o$, we have either of the following cases.

- The event o is a C-event. This means that it is justified by a C-event i. Additionally, it follows from Definition 6.22 and assumption $i \not\approx_v o$, that $i \not\approx_t o$. Since i and o are C-events, $i \curvearrowleft_v o$ implies $i \curvearrowleft_t o$. So we have $i \curvearrowleft_t o$ and $i \not\approx_t o$. Using (6.17), we conclude that $o \in \mathit{final}(t')$, and therefore $o \in \mathit{final}(v')$.

- The event o is an A-event. This means that it is justified by an event i which may either be an A-event or a C-event.

  – In the first case, by Definition 6.22, we have $i \not\approx_v o$ implies $i \not\approx_s o$. Since i and o are A-events, $i \curvearrowleft_v o$ implies $i \curvearrowleft_s o$. So we have $i \curvearrowleft_s o$ and $i \not\approx_s o$. Hence, using (6.15), we conclude that $o \in \mathit{final}(s')$ and therefore, $o \in \mathit{final}(v')$.

  – In the second case, it ensues that o is justified by a B-event b in s, which in turn, is justified by i in t; that is, $b \curvearrowleft_s o$ and $i \curvearrowleft_t b$. Moreover, we have by assumption $i \not\approx_v o$, which implies $i \not\approx_t b$ or $b \not\approx_s o$ (by Definition 6.22). To sum up, we have $b \curvearrowleft_s o$ and $i \curvearrowleft_t b$ and $(i \not\approx_t b$ or $b \not\approx_s o)$. This logically implies $(b \curvearrowleft_s o$ and $b \not\approx_s o)$ or $(i \curvearrowleft_t b$ and $i \not\approx_t b)$. In the first case, we use (6.15) and in the second, we use (6.17) to conclude that $o \in \mathit{final}(v')$.

For the second part, we assume that $\mathit{rest}(v) = \mathit{rest}(v')$ and $i \curvearrowleft_v o$ and $i \approx_v o$ and $i \in \mathit{final}(v')$ and show that $o \in \mathit{final}(v')$. Using Lemma 6.37, it follows that $\mathit{rest}(s) = \mathit{rest}(s')$ and $\mathit{rest}(t) = \mathit{rest}(t')$. From $i \curvearrowleft_v o$ and $i \approx_v o$ and $i \in \mathit{final}(v')$, we have either of the following cases.

- The event o is a C-event. This means that i is a C-event. Also, by Definition 6.22, it follows from assumption $i \approx_v o$ that $i \approx_t o$ and from assumption $i \in \mathit{final}(v')$ that $i \in \mathit{final}(t')$. So we have $i \curvearrowleft_t o$ and $i \approx_t o$ and $i \in \mathit{final}(t')$, and hence, using (6.18) we conclude that $o \in \mathit{final}(t')$, and therefore $o \in \mathit{final}(v')$.

- The event o is an A-event. This means that it is justified by an event i which may either be an A-event or a C-event.

  – In the first case, by Definition 6.22, $i \approx_v o$ implies $i \approx_s o$ and $i \in \mathit{final}(v')$ implies $i \in \mathit{final}(s')$. So we have $i \curvearrowleft_s o$ and $i \approx_s o$ and $i \in \mathit{final}(s')$. Using (6.16) we conclude that $o \in \mathit{final}(s')$, and therefore $o \in \mathit{final}(v')$.

– In the second case, it follows that o is justified by a B-event b in s, which in turn, is justified by i in t; that is, $b \curvearrowleft_s o$ and $i \curvearrowleft_t b$. Moreover, we have by assumption $i \approx_v o$, which implies $i \approx_t b$ and $b \approx_s o$ (by Definition 6.22). Additionally, we have $i \in \mathit{final}^i(v')$ which implies $i \in \mathit{final}^i(t')$. Using (6.18), we deduce that $b \in \mathit{final}^o(t')$. This implies that $b \in \mathit{final}^i(s')$ because s' and t' are assumed to be composable. Using (6.16), we deduce that $o \in \mathit{final}(s')$ and consequently, $o \in \mathit{final}(v')$. ■

We can now state the following result.

Theorem 6.39. *There is a lluf subcategory of* **CkProc** *consisting of deterministic processes and identity morphisms.*

The above theorem reflects the fact that identity morphisms are not observably deterministic according to our definition; see Example 6.40. Our characterisation of deterministic processes assumes that the notion of causality encoded by justification pointers is comprehensive; that is, the possibility of occurrence of an event only depends on the observation of its cause. However, this is not true in the case of the copycat strategy in Game Semantics. There is implicit causality information in the temporal order that is obfuscated by its round abstraction, the identity process. We further discuss this problem in the next section.

6.4 Discussion

We defined a way to extend the locally synchronous setting of Chapter 3 to global synchrony using a clock monad. Consequently, one can add global clocks to round abstracted models in a principled fashion. This operation is akin to mapping a hardware specification onto a physical medium. For example, while in a locally synchronous setting, the time elapsed between two successive rounds is unknown, it must be specified as clock ticks in a globally synchronous setting.

We saw that the clock monad provides a canonical way to wire processes with a global clock; however, it does not specify nor enforce the use of the clock. Hence, we defined a category of processes whose events synchronise on the global clock. We called such processes clocked.

We then explained how deterministic processes may be defined within the category of clocked processes. The definition of determinism in a synchronous setting has to take causality—expressed as justification pointers—into account. Ignoring it can lead to an erroneous characterisation of deterministic processes.

However, the importance of justification pointers extends beyond. If we reassign the pointers in a deterministic process in a legal manner, we may render the process nondeterministic. For example, consider the deterministic process $\sigma : A$ where $i_1 \vdash_A o_2$ and $i_2 \vdash_A o_3$, which only consists of the following two traces.

$$i_1 \cdot i_2 \cdot \langle\, i_1 \,,\, o_2 \,\rangle$$

$$i_1 \cdot i_2 \cdot \langle\, i_2 \,,\, o_3 \,\rangle$$

Another legal pointer assignment may be,

$$i_1 \cdot i_2 \cdot \langle\, i_1 \,,\, o_2 \,\rangle$$

$$i_1 \cdot i_2 \cdot \langle\, i_2 \,,\, o_3 \,\rangle$$

which renders σ nondeterministic.

Consequently, when working with low-level models lacking pointers, such as those used to represent synchronous hardware [Ghi07], we need to be able to derive the pointer causality from the structure of the traces in a unique and unambiguous way. Examples of this include the game semantic model of First-Order Idealized Algol [GM03] and the game semantics of Syntactic Control of Interference (SCI) [GS10].

Our discussion up to this point assumed that justification pointers encode sufficient causality. However, this assumption fails in the case of the identity process. It is thusly misidentified as nondeterministic.

To define determinism accurately, it is important to have a precise notion of causality. In our setting, the only form of causality is encoded by the justification pointers. However, these encode *necessary* rather than *sufficient* causality. As a result, some implicit causality information represented by the temporal ordering may be lost when moving from an asynchronous setting

to a synchronous one.

The ability to determine the causal order of events in a round is a focal problem in synchronous languages. One solution [BB91] is to view each round (called a macrostep) as consisting of many atomic microsteps which have an intrinsic causality order. In Reactive Modules [AH99], each round receives all inputs *before* producing outputs.

To illustrate the loss of information that may occur in round abstraction, consider the following example.

Example 6.40. Take the clocked identity over A where $L_A = \{a_1, a_2, a_3\}$ and $a_1 \vdash a_2, a_1 \vdash a_3$. It contains the following two traces.

$$s_1 = \langle\ tick\ ,\ a_1^i\ ,\ a_1^o\ \rangle \cdot \langle\ a_2^i\ ,\ a_2^o\ ,\ tick\ \rangle$$

$$s_2 = \langle\ tick\ ,\ a_1^i\ ,\ a_1^o\ \rangle \cdot \langle\ a_3^i\ ,\ a_3^o\ ,\ tick\ \rangle$$

In an asynchronous trace, there may be implicit causality information encoded by the temporal order of events which could be obfuscated when applying round abstraction. In our example, the occurrences of a_2^o and a_3^o cause the apparent nondeterminism. However, this conclusion does not account for the fact that copycat traces always take an input before producing its corresponding output copy. Since the justified synchronous identity is a round abstraction of copycat, the previous fact is obscured by making inputs simultaneous with their respective output copies. In this case, justification pointers alone do not encode sufficient causality information to accurately specify a deterministic process.

So, what can be done? We need traces that encode *comprehensive* causality information. In other words, for each event in a trace, we need to specify a set of events that must occur before or simultaneously with it. This causality information must be preserved in the round abstraction.

Possible ways of representing this causality include multiple justification pointers [Wal04] and partial orders [Win80, Jan08]. Both of these introduce challenges in the synchronous setting that must be addressed separately. For example, multiple justification pointers may introduce causality cycles in a round.

Finally, the two notions studied in this chapter, global synchrony and determinism, are essential facets of synchronous languages à la Esterel. The categories introduced here combine these two characteristics and may therefore offer a platform for studying connections between round abstraction and synchronous languages.

CHAPTER 7

CONCLUSION

"Et puis sans rien nier je repars à nouveau,
Je suis le point final d'un roman qui commence."
MALEK HADDAD

7.1 Summary

We examined the use of round abstraction in the adaptation of asynchronous trace models into the synchronous framework. Originally, round abstraction is a useful idea that forms an intrinsic part of the Reactive Modules language [AH99]. It was used in the context of model checking [AHR98, AW99], but its monolithic definition impinged upon its applicability to game models.

We began by introducing a trace model of low-level concurrency. The idea is to construct a basic category that allows modelling synchronous and asynchronous behaviour but avoids unnecessary restrictions.

In Chapter 4, we used our trace model to formulate and study the first compositional form of round abstraction. Unlike its precursor, our round abstraction does not hide intermediate events, and therefore, only strips some timing information. We defined two levels of abstraction: a partial one, requiring that all traces in the round abstraction stem from the original process; and a total round abstraction, additionally stipulating that all original traces be abstracted. For partial round abstraction, we identified sufficient conditions guaranteeing compositionality, yielding the concepts of compatibility and post-compatibility. Compositionality of total round abstraction is still an open question. We explored some pointers for further research.

Next, in Chapter 5, we extended the definition of synchronous process with justification pointers. The ability to assign an actual cause to each event in a trace is a prerequisite to describing asynchronous and deterministic behaviours. In certain game models, justification pointers can also be recovered from the structure of the trace [GM03]. However, this requires processes to satisfy stringent conditions; for example, restricting the game model to the first-order subset of a programming language. We chose to add pointers explicitly on traces in order to retain generality.

We then described a category of asynchronous processes. Intuitively, this category is structurally the same as the full subcategory of $\mathcal{G}$—the category of multi-threaded saturated strategies [GM08, pp. 14–18]—whose objects consist of questions only. Its processes simulate interleaved asynchrony by saturation under certain event permutations, a well-established idea in the literature [Udd86, JJH90, Lai01b, Lai05b, Lai06, GM08, Fos07]. We introduced new formalisations of the saturation preorder (Definition 5.31 and Definition 5.32), alternating copycat (Definition 5.36) and asynchronous identity (Definition 5.37); and presented previously unpublished categorical proofs (e.g. Lemma 5.51).

We ended Chapter 5 by showing that the compositionality of partial round abstraction holds in the causal and asynchronous categories. This result allows deriving sound synchronous representations of asynchronous processes, compositionally.

We demonstrated, in Chapter 6, that our locally synchronous processes can be lifted to a globally synchronous setting in a principled way. To this end, we introduced the ideas of *clock monoid* and *clock monad*. These categorical constructions allow us to extend our trace model with a global clock. Computational monads have been used, since Moggi [Mog89, Mog91], to extend categorical models with various effects but, to our knowledge, the notion of a clock monad is novel. It is important to note that clock monads describe how processes that have a global clock should be *wired*. However, it does not describe how the clock should be used. So, in the following section, we saw how a category of processes that synchronise on the clock can be constructed. Within this setting, we defined determinism on clocked processes and identified a subcategory of identity and deterministic processes.

We noted that despite our physical conceptions about the behaviour of the clocked identity, it is nondeterministic according to our definition. The problem is that clocked traces do not

contain sufficient causality information to allow an accurate characterisation of determinism. Indeed, justification pointers encode necessary rather than sufficient causality. We discussed how this problem may be overcome.

7.2 Further Directions

This thesis has established a compositional form of round abstraction and laid the foundation for the game-semantic analysis of synchrony. However, several aspects of our solution merit further study. There are also opportunities for new ideas to be built on this thesis. We explore some of these in the following points.

Generalisations. Compatibility and post-compatibility are sufficient to guarantee the compositionality of partial round abstraction. It would be interesting to see whether necessary conditions can be identified. Moreover, our setting reflects a low-level view of concurrency, where events are atomic and connectors do not implicitly buffer events. Eliminating these restrictions would lead to higher-level semantic models where round abstraction may reduce the cost of communication by sending larger 'packets' instead of individual events.

Total round abstraction. The question of compositionality of total round abstractions is an open problem. Immediately, there are two problems that seemingly hinder compositionality. The first one, illustrated in Example 4.15, is that round abstraction may introduce deadlock by aggregating mismatched events with those that compose well. The second is that round abstraction may introduce deadlock simply by assigning events to rounds in an inconsistent way (see Example 4.3). More generally, eliminating these problems may not be sufficient and further restrictions may be required. For instance, the notion of *interference* [Rey78]—a term given to the effect of the execution of a computation on another's outcome—seems of interest to total round abstraction. For example, the BSCI type system disallows functions from sharing identifiers—and therefore, from interfering—with their arguments. By contrast, pairs of arguments may interfere with each other. Intuitively, events from noninterfering computations may be allowed to occur simultaneously in the round abstraction since their order is arbitrary in any

original process [McC02]. However, making interfering events simultaneous is dangerous as it may result in either the round abstraction resolving or introducing deadlocks.

Synchronous game semantics. Synchronous representations of game models may be derived through partial round abstraction. However, in general, we are more interested in synchronous representations that fully capture the asynchronous behaviour of the original strategies. In this context, the compositionality of total round abstraction is important. This may be circumvented by designing a custom round abstraction that is total under certain conditions.

Geometry of Synthesis. GoS will, in principle, benefit from applying our results in the construction of provably correct compilers into synchronous circuits. The performance of the resulting circuits remains to be evaluated by practical case studies. Further optimisations at the level of circuitry should also be possible.

New synchronous languages. It is generally assumed that synchrony is too strong for practical use in high-level programming languages. One attempt at bridging this gap is Mandel and Pouzet's ReactiveML [MP05], which extends ML with synchronous primitives by adding processes and signals—entities that are orthogonal to the type system. By modelling processes as strategies and taking the start and end of computation as signals, it may be possible to design intrinsically synchronous programming languages. Imperative synchronous languages, like Esterel, already provide a rich set of primitives. Let us consider, for example, the statement await, which blocks until a signal is emitted. It may be given the type signature $com_1 \Rightarrow com_2 \times com_3$ and the following interpretation.

$$
\begin{array}{ccccc}
com_1 & \Rightarrow & com_2 & \times & com_3 \\
r_1 & & r_2 & & \\
d_1 & & d_2 & & \\
\hline
 & & & & r_3 \\
 & & r_2 & & d_3
\end{array}
$$

When activated (r_3), the strategy waits until com_2 is initiated, then terminates. Otherwise, the circuits behaves like the synchronous identity.

Connections with Interaction Categories. Interaction Categories [AGN96] are a general semantics framework for sequential and concurrent computation. They can express both synchronous and asynchronous behaviours, and have been used in conjunction with *specification structures* for guaranteeing certain properties, such as deadlock-freeness [AGN99]. Moreover, they have been shown to interpret Milner's SCCS calculus [AGN96] and model some synchronous programming languages [GN93]. Because of these reasons, they provide a general and flexible setting in which round abstraction should be expressed and studied.

Round abstraction and synchronous languages. Connections between round abstraction in general and synchronous languages should be investigated. As previously mentioned, Interaction Categories should provide a good platform for the study of round abstraction, and allow connections with Milner's SCCS and other synchronous languages to be established. On a more speculative note, round abstraction may be of use in the compilation of synchronous languages, either as an optimisation or a scheduling method. For instance, various Esterel compilers generate intermediate representations to which round abstraction can be applied; be it automata [BG92] or graphs [PBdS03].

Determinism on synchronous and clocked processes. Correctly defining determinism on synchronous processes is a nontrivial task. We demonstrated that a straightforward extension of the classical definition is incorrect. We then found that round abstraction may destroy information that is vital to discerning determinism. This led us to distinguish observable determinism and comprehensive determinism. The latter requires that justification is sufficiently encoded. It would be interesting to investigate the use of multiple justification and partial orders to overcome this problem.

REFERENCES

[AB84] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Science*, 30:91–131, 1984.

[Abr90] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Logic and Computation*, 1(1):5–41, 1990.

[Abr97a] S. Abramsky. Games in the semantics of programming languages. In P. Dekker, M. Stokhof, and Y. Venema, editors, *Proceedings of the 11th Amsterdam Colloquium*, pages 1–6. ILLC, Dept. of Philosophy, University of Amsterdam, 1997.

[Abr97b] S. Abramsky. Semantics of interaction: an introduction to game semantics. In P. Dybjer and A. Pitts, editors, *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute*, pages 1–31. Cambridge University Press, 1997.

[Abr01] S. Abramsky. Algorithmic game semantics: A tutorial introduction. In H. Schichtenberg and R. Steinbrüggen, editors, *Proceedings of the NATO Advanced Study Institute, Marktoberdorf*, chapter Proof and System Reliability, pages 21–47. Kluwer Academic Publishers, 2001.

[Abr10] S. Abramsky. From CSP to game semantics. In A. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 33–45. Springer, London, 2010.

[AGMO04] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying game semantics to compositional software modeling and verification. In K. Jensen and A. Podelski, editors, *TACAS '04: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Barcelona, Spain)*, volume 2988 of *Lecture Notes in Computer Science*, pages 421–435. Springer, 2004.

[AGN96] S. Abramsky, S. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In M. Broy, editor, *Proceedings of the 1994 Marktoberdorf Summer School on Deductive Program Design*, pages 35–113. Springer-Verlag, 1996.

[AGN99] S. Abramsky, S. J. Gay, and R. Nagarajan. A specification structure for deadlock-freedom of synchronous processes. *Theoretical Computer Science*, 222:1–53, 1999.

[AH89] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. In *LICS '89: Proceedings of the 4th Annual Symposium on Logic in Computer Science (Pacific Grove, California, USA)*, pages 138–145. IEEE Computer Society Press, 1989.

[AH94] L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115(2):179–247, 1994.

[AH95] R. Alur and T. A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In P. Wolper, editor, *CAV '95: Proceedings of the 7th International Conference on Computer Aided Verification (Liège, Belgium)*, volume 939 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 1995.

[AH97a] R. Alur and T. A. Henzinger. Modularity for timed and hybrid systems. In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory (Warsaw, Poland)*, volume 1243 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 1997.

[AH97b] R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. *International Journal on Software Tools for Technology Transfer*, 1(1–2):86–109, 1997.

[AH99] R. Alur and T. A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15:7–48, 1999.

[AHM98] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS '98: Proceedings of the 13th Annual Symposium on Logic in Computer Science (Indianapolis, Indiana, USA)*, pages 334–344. IEEE Computer Society, 1998.

[AHR98] R. Alur, T. A. Henzinger, and S. K. Rajamani. Symbolic exploration of transition hierarchies. In *TACAS' 98: Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 330–344. Springer, 1998.

[AJ92] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic (extended abstract). In R. K. Shyamasundar, editor, *FSTTCS '92: Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science (New Delhi, India)*, volume 652 of *Lecture Notes in Computer Science*, pages 291–301. Springer, 1992.

[AJ94] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *The Journal of Symbolic Logic*, 59(2):543–574, 1994.

[AJ05] S. Abramsky and R. Jagadeesan. A game semantics for generic polymorphism. *Annals of Pure and Applied Logic*, 133:3–37, 2005.

[AJ09] S. Abramsky and R. Jagadeesan. Game semantics for access control. *Electronic Notes in Theoretical Computer Science*, 249:135–156, 2009.

[AJM00] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 1991.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[AM94] S. Abramsky and G. McCusker. Games for recursive types. In C. Hankin, I. Mackie, and R. Nagarajan, editors, *Theory and Formal Methods*, pages 1–20. Imperial College Press, 1994.

[AM97a] S. Abramsky and G. McCusker. Call-by-value games. In M. Nielsen and W. Thomas, editors, *CSL '97: Proceedings of the 11th International Workshop on Computer Science Logic (Aarhus, Denmark)*, volume 1414 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1997.

[AM97b] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol. In P. O'Hearn and R. D. Tennent, editors, *Algol-like Languages*, pages 317–348. Birkhaüser, 1997.

[AM99a] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science*, 227:3–42, 1999.

[AM99b] S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, pages 1–56. Springer-Verlag, 1999.

[AM99c] S. Abramsky and P.-A. Melliès. Concurrent games and full completeness. In *LICS '99: Proceedings of the 14th Annual Symposium on Logic in Computer Science (Trento, Italy)*, pages 431–442. IEEE Computer Society Press, 1999.

[Ari28] Aristotle. *Topics*. Clarendon Press, Oxford, 1928.

[AW99] R. Alur and B.-Y. Wang. "Next" heuristic for on-the-fly model checking. In J. C. M. Baeten and S. Mauw, editors, *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory (Eindhoven, The Netherlands)*, volume 1664 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 1999.

[Bae05] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[BCE+03] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[BCG99] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In J. C. M. Baeten and S. Mauw, editors, *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory (Eindhoven, The Netherlands)*, volume 1664 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 1999.

[BCH+85] J. L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real-time data-flow language. In *Proceedings of the IEEE Real-Time Systems Symposium (San Diego, USA)*, San Diego, 1985.

[BDGL10] A. Bakewell, A. Dimovski, D. R. Ghica, and R. Lazic. Data-abstraction refinement: a game semantic approach. *International Journal on Software Tools for Technology Transfer*, 12(5):373–389, 2010.

[Ben01] A. Benveniste. Some synchronization issues when designing embedded systems from components. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT '01: Proceedings of the 1st International Workshop on Embedded Software (Tahoe City, California, USA)*, volume 2211 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2001.

[Ber99] G. Berry. The constructive semantics of pure Esterel. Draft book. Available at `http://www.esterel-technologies.com/`, 1999.

[BG88] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. Technical Report 842, INRIA-Sophia Antiopolis, 1988.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BG08] A. Bakewell and D. R. Ghica. On-the-fly techniques for game-based software model checking. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Budapest, Hungary)*, volume 4963 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2008.

[BG09] A. Bakewell and D. R. Ghica. Compositional predicate abstraction from game semantics. In S. Kowalewski and A. Philippou, editors, *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (York, UK)*, volume 5505 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2009.

[BKR+91] K. V. Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of the conference on European Design Automation*, pages 384–389. IEEE Computer Society Press, 1991.

[Bla92] A. Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56(1-3):183–220, 1992.

[BLJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[BMR83] G. Berry, S. Moisan, and J.-P. Rigault. Esterel: Towards a synchronous and semantically sound high level language for real time applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 30–40, 1983.

[Bou85] G. Boudol. Notes on algebraic calculi of processes. In *Logics and Models of Concurrent Systems*, pages 261–303. Springer, 1985.

[BRS93] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating Reactive Processes. In *POPL '93: Conference Record of the 20th Annual Symposium on Principles of Programming Languages (Charleston, South Carolina)*, pages 85–98. ACM, 1993.

[BS01] G. Berry and E. Sentovich. Multiclock esterel. In T. Margaria and T. F. Melham, editors, *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (Livingston, Scotland)*, volume 2144 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2001.

[Cas01] P. Caspi. Embedded control: From asynchrony to synchrony and back. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT '01: Proceedings of the 1st International Workshop on Embedded Software (Tahoe City, California, USA)*, volume 2211 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Conference Record of the 4th ACM Symposium on Principles of Programming Languages (Los Angeles, California, USA)*, pages 238–252. ACM, 1977.

[Cel] Celoxica Ltd. Handel-C language reference manual. `http://www.celoxica.com` available at `http://babbage.cs.qc.edu/courses/cs345/Manuals/HandelC.pdf`.

[Cha84] D. M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, 1984.

[CKP09] K. Chatzikokolakis, S. Knight, and P. Panangaden. Epistemic strategies and games on concurrent processes. In M. Nielsen, A. Kucera, P. B. Miltersen, C. Palamidessi, P. Tuma, and F. D. Valencia, editors, *SOFSEM '09: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (Spindleruv Mlýn, Czech Republic)*, volume 5404 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2009.

[CM10] A. C. C. Calderon and G. McCusker. Understanding game semantics through coherence spaces. In P. Selinger, editor, *MFPS XXVI: Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics (Ottawa, Ontario, Canada)*, number 265 in Electronic Notes in Theoretical Computer Science, pages 231–244. Elsevier, 2010.

[Con76] J. H. Conway. *On Numbers and Games*. Academic Press, London, 1976.

[De 85] R. De Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[DGL06] A. Dimovski, D. R. Ghica, and R. Lazic. A counterexample-guided refinement tool for open procedural programs. In A. Valmari, editor, *SPIN '06: Proceedings of the 13th International SPIN Workshop on Model Checking Software (Vienna, Austria)*, volume 3925 of *Lecture Notes in Computer Science*, pages 288–292. Springer, 2006.

[DH01] V. Danos and R. Harmer. The anatomy of innocence. In L. Fribourg, editor, *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic (Paris, France)*, volume 2142 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2001.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. EWD 123 available at `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`, September 1965.

[Dil88] D. L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. PhD thesis, Carnegie Mellon University, Feb. 1988.

[FHLR79] N. Francez, C. A. R. Hoare, D. J. Lehmann, and W. P. D. Roever. Semantics of nondeterminism, concurrency, and communication. *Journal of Computer and System Sciences*, 19:290–308, 1979.

[Fos07] L. Fossati. Handshake games. *Electronic Notes in Theoretical Computer Science*, 171(3):21–41, 2007.

[Gal95] D. R. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *FCCM '95: Proceedings of the 3rd IEEE Symposium on Field-Programmable Custom Computing Machines (Napa Valley, California, USA)*, pages 136–144. IEEE Computer Society, 1995.

[GB09] D. R. Ghica and A. Bakewell. Clipping: A semantics-directed syntactic approximation. In *LICS '09: Proceedings of the 24th Annual Symposium on Logic in Computer Science (Los Angeles, California, USA)*, pages 189–198. IEEE Computer Society, 2009.

[GGR94] U. Goltz, R. Gorrieri, and A. Rensink. On syntactic and semantic action refinement. In M. Hagiya and J. C. Mitchell, editors, *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software (Sendai, Japan)*, volume 789 of *Lecture Notes in Computer Science*, pages 385–404. Springer, 1994.

[Ghi07] D. R. Ghica. Geometry of Synthesis: A Structured Approach to VLSI Design. In *POPL '07: Proceedings of the 34th Annual Symposium on Principles of Programming Languages (Nice, France)*, pages 363–375. ACM Press, 2007.

[Ghi09a] D. R. Ghica. Applications of game semantics: From software analysis to hardware synthesis (invited tutorial paper). In *LICS '09: Proceedings of the 24th Annual Symposium on Logic in Computer Science (Los Angeles, California, USA)*, pages 17–26. IEEE Computer Society Press, 2009.

[Ghi09b] D. R. Ghica. Function interface models for hardware compilation: Types, signatures, protocols. *CoRR*, abs/0907.0749, 2009.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[GM] D. R. Ghica and M. N. Menaa. Synchronous game semantics via round abstraction. Submitted to the *Journal of the ACM*.

[GM03] D. R. Ghica and G. McCusker. The regular-language semantics of first-order Idealized Algol. *Theoretical Computer Science*, 309(1–3):469–502, 2003.

[GM06] D. R. Ghica and A. S. Murawski. Compositional model extraction for higher-order concurrent programs. In H. Hermanns and J. Palsberg, editors, *TACAS '06: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Vienna, Austria)*, volume 3920 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2006.

[GM08] D. R. Ghica and A. Murawski. Angelic semantics of fine-grained concurrency. *Annals of Pure and Applied Logic*, 151(2–3):89–114, 2008.

[GM10] D. R. Ghica and M. N. Menaa. On the compositionality of round abstraction. In P. Gastin and F. Laroussinie, editors, *CONCUR '10: Proceedings of the 21th International Conference on Concurrency Theory (Paris, France)*, volume 6269 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 2010.

[GM11] D. R. Ghica and M. N. Menaa. Synchronous game semantics via round abstraction. In M. Hofmann, editor, *FOSSACS '11: Proceedings of the 14th International Conference*

on Foundations of Software Science and Computational Structures (Saarbrücken, Germany), volume 6604 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2011.

[GMO06] D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. *Theoretical Computer Science*, 350(2–3):234–251, 2006.

[GMR10] G. Gößler, D. L. Métayer, and J.-B. Raclet. Causality analysis in contract violation. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors, *RV: Proceedings of the 1st International Conference on Runtime Verification (St. Julians, Malta)*, volume 6418 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2010.

[GN93] S. J. Gay and R. Nagarajan. Modelling SIGNAL in interaction categories. In G. L. Burn, S. J. Gay, and M. Ryan, editors, *Theory and Formal Methods*, Workshops in Computing, pages 148–158. Springer, 1993.

[GR01] R. Gorrieri and A. Rensink. Action refinement. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 1047–1146. Elsevier, 2001.

[GS10] D. R. Ghica and A. Smith. Geometry of Synthesis II: From games to delay-insensitive circuits. In P. Selinger, editor, *MFPS XXVI: Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics (Ottawa, Ontario, Canada)*, volume 265 of *Electronic Notes in Theoretical Computer Science*, pages 301–324, 2010.

[GS11] D. R. Ghica and A. Smith. Geometry of Synthesis III: Resource management through type inference. In T. Ball and M. Sagiv, editors, *POPL '11: Proceedings of the 38th Annual Symposium on Principles of Programming Languages (Austin, Texas, USA)*, pages 345–356. ACM, 2011.

[GSSAL94] R. Gawlick, R. Segala, J. F. Søgaard-Andersen, and N. A. Lynch. Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir, editors, *ICALP '94: Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 166–177. Springer, 1994.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Hal98] N. Halbwachs. Synchronous programming of reactive systems. In A. J. Hu and M. Y. Vardi, editors, *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification (Vancouver, BC, Canada)*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[Ham70] C. L. Hamblin. *Fallacies*. Methuen, London, 1970.

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. In A. L. Sangiovanni-Vincentelli and J. Sifakis, editors, *EMSOFT '02: Proceedings of the 2nd International Workshop on Embedded Software (Grenoble, France)*, volume 2491 of *Lecture Notes in Computer Science*, pages 240–251. Springer, 2002.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HL06] R. Harmer and O. Laurent. The anatomy of innocence revisited. In S. Arun-Kumar and N. Garg, editors, *FSTTCS '06: Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (Kolkata, India)*, volume 4337 of *Lecture Notes in Computer Science*, pages 224–235. Springer, 2006.

[HM99] R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In *LICS '99: Proceedings of the 14th Annual Symposium on Logic in Computer Science (Trento, Italy)*, pages 422–430. IEEE Computer Society Press, 1999.

[HM06] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 3–14, 2006.

[HO00] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, 2000.

[Hoa80] C. A. R. Hoare. A model for communicating sequential processes. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs*. Cambridge University Press, 1980.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. Available at `http://www.usingcsp.com/cspbook.pdf`.

[Hod04] W. Hodges. Logic and games. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, 2004. Also available at `http://plato.stanford.edu/entries/logic-games/`.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and models of concurrent systems*, volume 13 of *NATO ASI Series*, pages 477–498. Springer-Verlag, 1985.

[Hug97] D. J. D. Hughes. Games and definability for System F. In *LICS '97: Proceedings of the 12th Annual Symposium on Logic in Computer Science (Warsaw, Poland)*, pages 76–86. IEEE Computer Society Press, 1997.

[Hyl97] M. Hyland. Game semantics. In P. Dybjer and A. Pitts, editors, *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute*, pages 131–184. Cambridge University Press, 1997.

[JÖ2] J. Jürjens. Games in the semantics of programming languages. *Synthese*, 133(1–2):101–120, October/November 2002.

[Jac94] B. Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69(1):73–106, 1994.

[Jan08] R. Janicki. Relational structures model of concurrency. *Acta Informatica*, 45(4):279–320, 2008.

[JJH90] H. Jifeng, M. B. Josephs, and C. A. R. Hoare. A theory of synchrony and asynchrony. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods*, pages 459—478. Elsevier, 1990.

[Joy77] A. Joyal. Remarques sur la théorie des jeux à deux personnes. *Gazette des Sciences Mathématiques du Quebec*, 1(4), 1977.

[JS93] A. Joyal and R. Street. Braided tensor categories. *Advances in Mathematics Advances in Mathematics*, 102:20–78, 1993.

[KD90] D. Ku and G. DeMicheli. HardwareC – a language for hardware design version 2.0. Technical Report CSL-TR-90-419, Stanford University, 1990.

[Koc70] A. Kock. Monads on symmetric monoidal closed categories. *Archiv der Mathematik*, 21:1–10, 1970.

[Koc72] A. Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23(1):113–120, 1972.

[Lai97] J. Laird. Full abstraction for functional languages with control. In *LICS '97: Proceedings of the 12th Annual Symposium on Logic in Computer Science (Warsaw, Poland)*, pages 58–67. IEEE Computer Society Press, 1997.

[Lai01a] J. Laird. A fully abstract games semantics of local exceptions. In *LICS '01: Proceedings of the 16th Annual Symposium on Logic in Computer Science (Boston, Massachusetts, USA)*, pages 105–114, 2001.

[Lai01b] J. Laird. A game semantics of Idealized CSP. In *MFPS XVII: Proceedings of the 17th Conference on the Mathematical Foundations of Programming Semantics (Aarhus, Denmark)*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 1–26, 2001.

[Lai03] J. Laird. A game semantics of linearly used continuations. In A. D. Gordon, editor, *FOSSACS '03: Proceedings of the 6th International Conference on Foundations of Software Science and Computational Structures (Warsaw, Poland)*, volume 2620 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2003.

[Lai05a] J. Laird. Decidability in syntactic control of interference. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *ICALP '05: Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (Lisbon, Portugal)*, volume 3580 of *Lecture Notes in Computer Science*, pages 904–916. Springer, 2005.

[Lai05b] J. Laird. A game semantics of the asynchronous π-calculus. In M. Abadi and L. de Alfaro, editors, *CONCUR '05: Proceedings of the 16th International Conference on Concurrency Theory (San Francisco, CA, USA)*, volume 3653 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2005.

[Lai06] J. Laird. Game semantics for higher-order concurrency. In S. Arun-Kumar and N. Garg, editors, *FSTTCS '06: Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (Kolkata, India)*, volume 4337 of *Lecture Notes in Computer Science*, pages 417–428. Springer, 2006.

[Lai07] J. Laird. A fully abstract trace semantics for general references. In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *ICALP '07: Proceedings of the 34nd International Colloquium on Automata, Languages and Programming (Wroclaw, Poland)*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007.

[LGAD04] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowksi. Semantics and implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, 5(3):182–199, 2004.

[LGBBG86] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal: A data flow-oriented language for signal processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(2):362–374, 1986.

[LGG+08] F. Loulergue, F. Gava, L. Gesbert, G. Hains, and J. Tesson. *Bulk Synchronous Parallel ML 0.4 beta Reference Manual*, October 2008. Available at `http://bsmllib.free.fr/`.

[LHF00] F. Loulergue, G. Hains, and C. Foisy. A calculus of functional BSP programs. *Science of Computer Programming*, 37:253–277, 2000.

[LL78] P. Lorenzen and K. Lorenz. *Dialogische Logik*. Wissenschaftliche Buchgesellschaft, Darmstadt, Germany, 1978.

[LLGL91] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[Lor60] P. Lorenzen. Logik und agon. *Atti del Congresso Internazionale di Filosofia*, 4:187–194, 1960.

[Lor61] P. Lorenzen. Ein dialogisches konstruktivitätskriterium. In *Infinitistic Methods (PWN, Warsaw, 1959)*, pages 193–200, Warsaw, 1961. Pergamon Press.

[Lor68] K. Lorenz. Dialogspiele als semantische grundlage von logikkalkülen. *Archiv für mathematische Logik und Grundlagenforschung*, 11(3):73–100, 1968.

[Lor01] K. Lorenz. Basic objectives of dialogue logic in historical perspective. *Synthese*, 127(1):255–263, 2001.

[LT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LTL03] P. Le Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003.

[Mac98] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1998.

[Mar86] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[Mar87] A. J. Martin. A synthesis method for self-timed VLSI circuits. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 224–229. IEEE Computer Society Press, 1987.

[Maz77] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Tech. Rep. PB-78, Aarhus University, 1977.

[Maz95] A. Mazurkiewicz. Introduction to trace theory. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*. World Scientific Books, 1995.

[McC97] G. McCusker. Games and definability for fpc. *Bulletin of Symbolic Logic*, 3(3):347–362, September 1997.

[McC02] G. McCusker. A fully abstract relational model of Syntactic Control of Interference. In J. C. Bradfield, editor, *CSL '02: Proceedings of the 16th International Workshop on Computer Science Logic (Edinburgh, Scotland)*, volume 2471 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2002.

[McC07] G. McCusker. Categorical models of syntactic control of interference revisited, revisited. *LMS Journal of Computation and Mathematics*, 10:176–206, 2007.

[McC10] G. McCusker. A graph model for imperative computation. *Logical Methods in Computer Science*, 6(1):1–35, 2010.

[Mel06] P.-A. Melliès. Asynchronous games 2: The true concurrency of innocence. *Theoretical Computer Science*, 358(2-3):200–228, 2006.

[Men09] M. N. Menaa. Towards a synchronous game semantics. Slides presented at the 4th Workshop on Games for Logic and Programming Languages (GaLoP IV). Available at `http://www.comlab.ox.ac.uk/galop09/`, 2009.

[Men10] M. N. Menaa. On the compositionality of round abstraction. Short paper presented at the 25th Annual Symposium on Logic in Computer Science (LICS '10), Edinburgh, Scotland, 2010.

[Mil75] R. Milner. Processes: A mathematical model of computing agents. In H. Rose and J. Shepherdson, editors, *Proceedings of the Logic Colloquium '73,*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 157–173. Elsevier, 1975.

[Mil77] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[MM07] P.-A. Melliès and S. Mimram. Asynchronous games: Innocence without alternation. In L. Caires and V. T. Vasconcelos, editors, *CONCUR '07: Proceedings of the 18th International Conference on Concurrency Theory (Lisbon, Portugal)*, volume 4703 of *Lecture Notes in Computer Science*, pages 395–411. Springer, 2007.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *LICS '89: Proceedings of the 4th Annual Symposium on Logic in Computer Science (Pacific Grove, California, USA)*, pages 14–23. IEEE Computer Society, 1989.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[MP05] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *PPDR '05: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (Lisbon, Portugal)*, pages 82–93. ACM, 2005.

[Mur10] A. S. Murawski. Full abstraction without synchronization primitives. In P. Selinger, editor, *MFPS XXVI: Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics (Ottawa, Ontario, Canada)*, number 265 in Electronic Notes in Theoretical Computer Science, pages 423–436. Elsevier, 2010.

[Nic94] H. Nickau. Hereditarily sequential functionals. In A. Nerode and Y. Matiyasevich, editors, *LFCS '94: Proceedings of the 3rd International Symposium on Logical Foundations of Computer Science (St. Petersburg, Russia)*, volume 813 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 1994.

[NSW93] M. Nielsen, V. Sassone, and G. Winskel. Relationships between models of concurrency. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX School/Symposium: A Decade of Concurrency, Reflections and Perspectives (Noordwijkerhout, The Netherlands)*, volume 803 of *Lecture Notes in Computer Science*, pages 425–476. Springer, 1993.

[NV07] S. Nain and M. Y. Vardi. Branching vs. linear time: Semantical perspective. *Automated Technology for Verification and Analysis*, pages 19–34, 2007.

[NV09] S. Nain and M. Y. Vardi. Trace semantics is fully abstract. In *LICS '09: Proceedings of the 24th Annual Symposium on Logic in Computer Science (Los Angeles, California, USA)*, pages 59–68. IEEE Computer Society, 2009.

[NvRS88] C. Niessen, C. H. van Berkel, M. Rem, and R. W. J. J. Saeijs. VLSI programming and silicon compilation; a novel approach from Philips Research. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 150–151. IEEE Computer Society Press, 1988.

[OBL98] J. O'Leary, G. Brown, and W. Luk. Verified compilation of communicating processes into clocked circuits. *Formal Aspects of Computing*, 9(5-6):537–559, 1998.

[O'H03] P. W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.

[OPTT99] P. W. O'Hearn, J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228(1-2):211–252, 1999.

[Pag96] I. Page. Constructing hardware-software systems from a single description. *VLSI Signal Processing*, 12(1):87–107, 1996.

[PBC07] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundamenta Informaticae*, 78(1):131–159, 2007.

[PBdS03] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of esterel programs. In *MEMOCODE '03: Proceedings of the 1st ACM & IEEE International Conference on Formal Methods and Models for Co-Design (Mont Saint-Michel, France)*, pages 227–236. IEEE Computer Society, 2003.

[PBEB07] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling ESTEREL*. Springer, 2007.

[Pet62] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proceedings of the IFIP Congress*, pages 386–390. North-Holland, Amsterdam, 1962.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[PPE+97] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51–57, 1997.

[Red96] U. S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *Lisp and Symbolic Computation*, 9(1):7–76, 1996.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *POPL '78: Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages (Tucson, Arizona, USA)*, pages 39–46. ACM, 1978.

[Rey81] J. C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345—372. North-Holland, 1981.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, London, 1998.

[Sch00] S. Schneider. *Concurrent and Real-time Systems: the CSP Approach*. Worldwide Series in Computer Science. Wiley, Chichester, 2000.

[SCME06] C. Stetson, X. Cui, P. R. Montague, and D. M. Eagleman. Motor-sensory recalibration leads to an illusory reversal of action and sensation. *Neuron*, 51(5):651–659, September 2006.

[SSC01] G. Snider, B. Shackleford, and R. J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *FPGA '01: Proceedings of the 9th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, California, USA)*, pages 115–124. ACM, 2001.

[Sut89] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[SW01] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *CASES: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (Atlanta, Georgia, USA)*, pages 49–58. ACM, 2001.

[TdS04] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in Esterel. In *MEMOCODE '04: Proceedings of the 2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (San Diego, California, USA)*, pages 39–48. IEEE, 2004.

[TvD88] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: an Introduction*. North-Holland, 1988.

[Udd86] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1(4):197–204, 1986.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, 1990.

[vS88] C. H. van Berkel and R. W. J. J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 157–162. IEEE Computer Society Press, 1988.

[Wal04] M. Wall. *Games for Syntactic Control of Interference*. PhD thesis, University of Sussex, 2004.

[Win80] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[Win87] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1987.

[Wir98] N. Wirth. Hardware compilation: Translating programs into circuits. *IEEE Computer*, 31(6):25–31, 1998.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Vol. 4*, pages 1–148. Oxford University Press, 1995.

[YH98] H. Yang and H. Huang. Type reconstruction for syntactic control of interference, part 2. In *ICCL '98: Proceedings of the International Conference on Computer Languages (Chicago, Illinois, USA)*, pages 164–173. IEEE Computer Society, 1998.

[ZRM90] Y. Zhang, J. Roivainen, and A. Mämmelä. Clock-gating in FPGAs: A novel and comparative evaluation. In *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 584–590.