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
This unpublished thesis/dissertation is copyright of the author and/or third parties.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.
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
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A landscape of relevant literature</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>A subset of CSP syntax</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The trace semantics of CSP</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Asynchronous product versus synchronous product of automata</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Some Esterel statements</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>A cyclic Esterel program</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>The arena of natural numbers nat</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>The product arena of $A$ and $B$</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>The function arena of $A$ and $B$</td>
<td>21</td>
</tr>
<tr>
<td>2.10</td>
<td>The addition of 5 and 7 produced by composition</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Coherence condition for $\alpha$</td>
<td>61</td>
</tr>
<tr>
<td>3.2</td>
<td>Coherence condition for $\lambda$ and $\rho$</td>
<td>61</td>
</tr>
<tr>
<td>3.3</td>
<td>Coherence conditions for $\gamma$</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>A Counter in Reactive Modules</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>A round abstracted Counter</td>
<td>69</td>
</tr>
<tr>
<td>4.3</td>
<td>Round abstraction on automata</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>When compatibility is too strong</td>
<td>74</td>
</tr>
<tr>
<td>4.5</td>
<td>Totality of $\preceq_u$ in Lemma 4.10</td>
<td>78</td>
</tr>
<tr>
<td>5.1</td>
<td>Lemma 5.16 Case 2</td>
<td>93</td>
</tr>
<tr>
<td>5.2</td>
<td>Lemma 5.16 Case 3</td>
<td>93</td>
</tr>
<tr>
<td>5.3</td>
<td>Pointer structure inheritance in Lemma 5.20</td>
<td>96</td>
</tr>
<tr>
<td>5.4</td>
<td>Causality in $id_{A \otimes B}$</td>
<td>98</td>
</tr>
<tr>
<td>6.1</td>
<td>Clock monad</td>
<td>134</td>
</tr>
<tr>
<td>6.2</td>
<td>Justification pointers in traces $u \in a_{Ck,Ck,Ck};id_{Ck} \otimes sp_{Ck};sp_{Ck}$ in Lemma 6.4</td>
<td>142</td>
</tr>
<tr>
<td>6.3</td>
<td>Justification pointers in traces $v \in sp_{Ck} \otimes id_{Ck};sp_{Ck}$ in Lemma 6.4</td>
<td>142</td>
</tr>
<tr>
<td>6.4</td>
<td>Justification pointers in traces $u \in cis_{Ck} \otimes id_{Ck};sp_{Ck}$ in Lemma 6.4</td>
<td>142</td>
</tr>
<tr>
<td>6.5</td>
<td>Justification pointers in traces $v \in \lambda_{Ck}$ in Lemma 6.4</td>
<td>142</td>
</tr>
</tbody>
</table>
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 \textit{SynProc}_T .................. 152
6.9 A circuit representation of \((id_B \otimes T) \circ T \text{eval}_{B,C_T} \) ......... 152
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.
CHAPTER 1. INTRODUCTION

“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 [Pl77]. 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
1.2. OUTLINE AND CONTRIBUTIONS

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.


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


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.

<table>
<thead>
<tr>
<th>Process calculi</th>
<th>Asynchronous</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP [Hoa85]</td>
<td>CRP [BRS93], SCCS [Mil83]</td>
<td></td>
</tr>
<tr>
<td>Programming languages</td>
<td>ICA [CM08], BSCI [Rey78]</td>
<td>Esterel [BMR83], ReactiveML [MP05]</td>
</tr>
<tr>
<td>Games</td>
<td>Hyland-Ong [HO00], [GS11]</td>
<td>Clocked circuits [Ghi07]</td>
</tr>
<tr>
<td>Hardware</td>
<td>Event logic [Sut89]</td>
<td></td>
</tr>
</tbody>
</table>

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 [CM08]. 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.
2.1. SYNCHRONY AND ASYNCHRONY

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.

\[\text{\footnotesize \text{This limited form of synchrony is often called rendezvous or handshake communication.}}\]
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 $P \Box Q$ 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 $SKIP; P = P$

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 $P \Box Q$ is trace equivalent to $P \cap Q$, 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 $\epsilon$. 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, $\text{traces}(P)$, is

- **nonempty:** it always contains the empty trace $\epsilon$ 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 $\checkmark$.

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

```
\begin{align*}
\text{traces}(\text{STOP}) &= \{\epsilon\} \\
\text{traces}(&\text{SKIP}) = \{\epsilon, \checkmark\} \\
\text{traces}(&a \rightarrow P) = \{a \cdot s \mid s \in \text{traces}(P)\} \\
\text{traces}(P \setminus X) &= \{s \mid (aP - X) \mid s \in \text{traces}(P)\} \\
\text{traces}(P \Box Q) &= \text{traces}(P) \cup \text{traces}(Q) \\
\text{traces}(P \cap Q) &= \text{traces}(P) \cap \text{traces}(Q) \\
\text{traces}(P \parallel Q) &= \text{traces}(P) \cap \text{traces}(Q) \\
\text{traces}(P \mid\mid Q) &= \{s \mid s \upharpoonright aP \in \text{traces}(P) \text{ and } s \upharpoonright aQ \in \text{traces}(Q)\} \\
\text{traces}(P ; Q) &= \text{traces}(P) \cup \{s \cdot \checkmark \cdot t \mid s \cdot \checkmark \in \text{traces}(P) \text{ and } t \in \text{traces}(Q)\}
\end{align*}
```

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) \[\text{[Mil83]}\] 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 $\text{Act}$ of atomic actions: time is assumed to be discrete and actions indivisible in time. For each action $a \in \text{Act}$ 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 \[\text{[Cas01]}\], contrasts this synchronous product with its asynchronous equivalent. Idleness can be modelled using a reserved action ‘1’ and further requires that $(\text{Act}, \times, 1)$ is an Abelian monoid.

![Figure 2.4: Asynchronous product versus synchronous product of automata](image-url)
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 [PBE87].

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.
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 \cite{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 \ref{fig:causal} 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 \cite{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 \cite{TvD88}. For more on the theory of causality, refer to \cite{Ber99}.

Figure 2.5: Some Esterel statements

<table>
<thead>
<tr>
<th>Esterel Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nothing</td>
<td>no-op</td>
</tr>
<tr>
<td>pause</td>
<td>hold until the next instant</td>
</tr>
<tr>
<td>p; q</td>
<td>sequential composition</td>
</tr>
<tr>
<td>p</td>
<td></td>
</tr>
<tr>
<td>loop p end</td>
<td>infinite iteration</td>
</tr>
<tr>
<td>emit S</td>
<td>emit signal S</td>
</tr>
<tr>
<td>present S then p else q end</td>
<td>signal branching</td>
</tr>
<tr>
<td>suspend p when S</td>
<td>pre-emption test</td>
</tr>
<tr>
<td>if v then p else q end</td>
<td>conventional branching</td>
</tr>
</tbody>
</table>
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 [BCC99].

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 Aristo-
tle [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 pro-
tagonsists: 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 [ACMO04, 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, HyI97, MC97, 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).

<table>
<thead>
<tr>
<th>Move</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O question</td>
<td>request output from P</td>
</tr>
<tr>
<td>O answer</td>
<td>provide input to P</td>
</tr>
<tr>
<td>P question</td>
<td>request input from O</td>
</tr>
<tr>
<td>P answer</td>
<td>provide output to O</td>
</tr>
</tbody>
</table>

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.

\[
\begin{align*}
q^P \\
q^O \leftarrow q'^O \\
q^O \leftarrow q^P \rightarrow a^O \\
q^O \leftarrow q^P \rightarrow a^O
\end{align*}
\]

With the concepts introduced so far, we can already give interesting examples. The arena \( \text{nat} \) 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.

\[\text{nat} \]
\[\begin{align*}
O & \quad q \\
P & \quad 5
\end{align*}\]

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 \text{nat}](image)
2.2. GAME SEMANTICS

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

$$\begin{array}{c|c|c|c|c|c}
\text{nat} \times \text{nat} & \text{nat} \times \text{nat} \\
O & q & O & q \\
P & 5 & P & 7 \\
O & q & O & q \\
P & 7 & P & 5 \\
\end{array}$$

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.

$$\begin{array}{c|c|c|c|c|c}
\text{nat} \times \text{nat} \Rightarrow \text{nat} \\
O & q \\
P & q \\
O & n \\
P & q \\
O & m \\
P & n + m \\
\end{array}$$
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.

\[
\begin{array}{c}
\text{nat} \Rightarrow \text{nat} \\
O & q \\
P & q \\
O & q \\
P & q \\
O & q \\
P & 1 \\
O & n \\
P & n \\
O & m \\
P & m \\
\end{array}
\]

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 $((\text{nat}_1 \Rightarrow \text{nat}_2) \Rightarrow \text{nat}_3) \Rightarrow \text{nat}_4$. While $M$ has plays of the form,

\[
\begin{array}{c}
((\text{nat}_1 \Rightarrow \text{nat}_2) \Rightarrow \text{nat}_3) \Rightarrow \text{nat}_4 \\
O \\
P \\
O \\
P \\
O \\
P \\
\end{array}
\]
the second term, \( N \), has plays of the following shape.

\[
((\text{nat}_1 \Rightarrow \text{nat}_2) \Rightarrow \text{nat}_3) \Rightarrow \text{nat}_4
\]

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 \( \sigma, \tau \) 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 \text{nat} \!, for example, we get the following play.

\[
\text{nat} \Rightarrow \text{nat}
\]

\[
\begin{array}{c}
O & q \\
P & q \\
O & n \\
P & n
\end{array}
\]
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 \( \pi \)-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, JH90]. 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 \(\preceq\), 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 \(\preceq_0\) on \(P_A\) satisfying, for all \(s, s' \in P_A\), we have \(s' \preceq_0 s\) 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 \(\preceq\) on \(P_A\) as the least reflexive and transitive relation containing \(\preceq_0\).

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 \(\preceq\). 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 princi-
pally 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 rea-
sons for this is poor support for concurrency in general-purpose programming languages. This
so-called semantic gap [SSC01] focused efforts on extending conventional programming lan-
guages 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 be-
coming 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 con-
trol 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 \textit{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 \textit{activation managers} in the context of hardware compilation [Ghi09b].

\subsection{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 \[McC02\].
2.3. HARDWARE COMPILATION VIA GEOMETRY OF SYNTHESIS

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 \[ \text{[GMO06]} \]. This makes it possible to automatically verify BSCI programs using conventional finite-state model checking techniques \[ \text{[GM06]} \]. 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,

\[
\sigma ::= \text{com} \mid \text{exp} \mid \text{var}
\]

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.

\[
\theta ::= \sigma \mid \theta \times \theta' \mid \theta \rightarrow \theta'
\]

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.

\[
\begin{align*}
\Gamma, x : A & \vdash x : A & \text{Axiom} \\
\Gamma, x : A & \vdash M : B & \\
\Gamma & \vdash \lambda x : A. M : A \rightarrow B & \text{Abstraction} \\
\Delta & \vdash M : A \rightarrow B & \\
\Gamma, \Delta & \vdash N : A & \\
\Gamma & \vdash MN : B & \text{Application} \\
\Gamma & \vdash M : A & \\
\Gamma & \vdash N : B & \\
\Gamma & \vdash \langle M, N \rangle : A \times B & \text{Product} \\
\Gamma, x : A & \vdash M : B & \\
\Gamma, x : A & \vdash M : B & \text{Weakening}
\end{align*}
\]

Observe that the application rule above requires \(\Gamma\) and \(\Delta\) to be disjoint.
The imperative part of BSCI is given by the following constants.

\[
\begin{align*}
\text{skip} : & \text{com} \quad \text{no-op} \\
\text{n} : & \text{exp} \quad \text{natural number constants} \\
\odot : & \text{exp} \times \text{exp} \rightarrow \text{exp} \quad \text{arithmetic and arithmetic-logic operators} \\
\& : & \text{com} \times A \rightarrow A \quad \text{sequential composition}, A \in \{\text{com, nat, var}\} \\
\parallel : & \text{com} \rightarrow \text{com} \rightarrow \text{com} \quad \text{parallel command composition} \\
:= : & \text{var} \times \text{exp} \rightarrow \text{com} \quad \text{assignment} \\
! : & \text{var} \rightarrow \text{exp} \quad \text{dereferencing} \\
\text{if} : & \text{exp} \times A \times A \rightarrow A \quad \text{selection}, A \in \{\text{com, exp, var}\} \\
\text{while} : & \text{exp} \times \text{com} \rightarrow \text{com} \quad \text{iteration} \\
\text{newvar} : & (\text{var} \rightarrow A) \rightarrow A \quad \text{local variable}, A \in \{\text{com, exp}\}.
\end{align*}
\]

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 \textit{Syntactic Control of Concurrency} (SCC), which allows shared-memory concurrency \cite{GMO06}. Moreover, it was recently shown that almost any recursion-free Idealized Concurrent Algol programs \cite{GM08}, barring pathological examples, can be automatically ‘serialised’ into BSCI via SCC \cite{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 \cite{Ghi07,McC02,McC10}.
2.3. HARDWARE COMPILATION VIA GEOMETRY OF SYNTHESIS

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 $\mathbf{HC}$ 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 $\lambda_A^{io}$ and $\lambda_A^{ra}$ for the composition of $\lambda_A$ with the first and second projections respectively. Morphisms $f : A \to B$ are circuits with the following interface.

\[
L_{A\to B} = L_A + L_B \\
\lambda_{A\to B} = [\langle \lambda_A^{io}, \lambda_A^{ra}\rangle, \lambda_B]
\]

Note that the input/output polarity of the $A$ ports is reversed while their request/acknowledgement polarity remains unchanged.
Given circuits \( f : A \rightarrow B \) and \( g : B \rightarrow C \), their composition is the circuit \( f;g : A \rightarrow C \) 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 \( \text{id} : A \rightarrow A \) 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, \( f \otimes g : A \otimes C \rightarrow B \otimes D \) 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 [HC00].

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.

- $[/com] = \{ R \mapsto (i,r), D \mapsto (a,a) \}$
- $[/exp] = \{ Q \mapsto (i,r), T \mapsto (a,a), F \mapsto (o,a) \}$
- $[/var] = \{ WT \mapsto (i,r), WF \mapsto (i,r), Q \mapsto (i,r), D \mapsto (a,a), T \mapsto (o,a), F \mapsto (o,a) \}$
where \( R \) stands for ‘run’, \( D \) for ‘done’ and \( W \) for ‘write’; and \( I_{\text{com}} = \{ R \} \), \( I_{\text{exp}} = \{ Q \} \), \( I_{\text{var}} = \{ WT, WF, Q \} \). For product and function types we have,

\[
\begin{align*}
\llbracket \theta \times \theta' \rrbracket &= \llbracket \theta \rrbracket \times \llbracket \theta' \rrbracket \\
\llbracket \theta \Rightarrow \theta' \rrbracket &= \llbracket \theta \rrbracket \Rightarrow \llbracket \theta' \rrbracket.
\end{align*}
\]

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

\[
\bigotimes_{1 \leq i \leq n} \llbracket A_i \rrbracket_s \xrightarrow{\llbracket x_1 : A_1, \ldots, x_n : A_n \vdash M : A \rrbracket}_s \llbracket A \rrbracket.
\]

The imperative constants of the language are interpreted using morphisms in \( \text{SHC} \). 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).
2.3. HARDWARE COMPILATION VIA GEOMETRY OF SYNTHESIS

\[ \text{\( J : \var_1 \times \exp_2 \rightarrow \com_3 \)} \]

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).

\[ \text{\([! : \var_1 \rightarrow \exp_2]\)} \]

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).

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.

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.

\[
\begin{align*}
\llbracket x : \theta \vdash x : \theta \rrbracket &= id[\theta] \\
\llbracket \Gamma, x : \theta \vdash x : \theta \rrbracket &= ! \otimes id[\theta] \\
\llbracket \Gamma \vdash \lambda x. M : \theta' \to \theta \rrbracket &= \Lambda(\llbracket \Gamma, x : \theta' \vdash M : \theta \rrbracket)
\end{align*}
\]
A soundness theorem is given, which is also a correctness result for the compilation technique.

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

### 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 $\text{SProc}$ [[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.
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
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.
- $\pi_A : L_A \rightarrow \{i, o\}$ 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 $\pi^*$ for a labelling function that is like $\pi$, 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.

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

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

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. TRACES AND PROCESSES

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 \(\lambda_s : E_s \rightarrow L_A\) 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 \(a.b.(a,c)\). 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 \((E, \preceq, \lambda)\) 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 \(s = (E_s, \preceq_s, \lambda_s)\) and \(t = (E_t, \preceq_t, \lambda_t)\) are equivalent if there exists a bijection \(\phi : E_s \to E_t\) satisfying \((\forall e, e' \in E_s)(e \preceq_s e' \iff \phi(e) \preceq_t \phi(e'))\) 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, \(\preceq_s \subseteq \preceq_t\). 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
3.1. TRACES AND PROCESSES

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 $s = (E_s, \leq_s, \lambda_s)$, $t = (E_t, \leq_t, \lambda_t)$, denoted by $s \cdot t$, is the trace defined by the triple $(E_s + E_t, \leq_s + \leq_t + (E_s \times E_t), [\lambda_s, \lambda_t])$.

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, $\text{last}(s) = \{e \in E_s \mid (\forall e' \in E_s)(e' \leq_s e)\}$. 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 $(E_s + E_t, \leq', [\lambda_s, \lambda_t])$, where $\leq' = \leq_s + \leq_t + E_s \times E_t + \text{first}(t) \times \text{last}(s)$.

3.1.3 Processes

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

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

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

For any signatures $A, B$ and $C$ we define the set of interaction traces, written $\text{int}(A, B, C)$, as $\Theta((A \Rightarrow B) \Rightarrow C)$. Given another signature $D$, let $\text{int}(A, B, C, D)$ 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 $\sigma \upharpoonright \tau = \{ u \in \text{int}(A, B, C) \mid u \upharpoonright A + B \in \sigma \text{ and } u \upharpoonright B + C \in \tau \}.$

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

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 $\text{SynProc}.$

### 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 $\text{SynProc}$ of processes has

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

$$
\sigma : A \to B \quad \tau : B \to C
\quad \sigma ; \tau : A \to C
$$

- 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 $|\lambda_u(e) = \text{inl}(a)|$ if and only if $\lambda_u(e') = \text{inr}(a)$ and $|\lambda_u(e) = \text{inr}(a)|$ if and only if $\lambda_u(e') = \text{inl}(a),$ where $a \in L_A.
We will need the following definitions.

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

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

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

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* $\text{out} : \Theta(A + B) \rightarrow \Theta(A)$ *such that for every* $t = \langle E, \preceq, \lambda \rangle$ *in* $\Theta(A + B)$

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

*where*

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

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

We begin by showing that $\text{SynProc}$ *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 $\text{in}(s) = \langle E, \preceq, \text{lin} \circ \lambda \rangle$, *where* $\text{lin} : L_A \rightarrow L_A + L_B$, *be an injection*. If $e, e' \in E$ and $e \approx e'$ *and* $(\text{lin} \circ \lambda)(e) = (\text{lin} \circ \lambda)(e')$, *then* $\lambda(e) = \lambda(e')$ *because* $\text{lin}$ *is injective*. It follows that $e = e'$ *because* $s$ *is a trace.* ■
Lemma 3.19. **Projection preserves the temporal order**. That is, if \( s \) is a trace over \( A + B \) and \( e, e' \) \( \in \) \( E_s \) such that \( \lambda_s(e) = \lambda_s(e') \in \text{lin}(L_A) \) and \( e \preceq_s e' \), then \( e \preceq_{s|A} e' \).

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

Lemma 3.20. **Projection on traces preserves singularity.**

**Proof.** Let \( s \) be a trace over \( A + B \). Let \( s \upharpoonright A = \langle E_0 = \{ e \in E_s | \lambda_s(e) \in \text{lin}(L_A) \}, \preceq_0 = \preceq_s \cap (E_0 \times E_0), \lambdaout \circ \lambda_s \rangle \), where \( \lambdaout : L_A + L_B \rightharpoonup L_A \) 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 \( (\lambdaout \circ \lambda_s)(e) = (\lambdaout \circ \lambda_s)(e') \), then \( \lambda_s(e) = \lambda_s(e') \) because \( \lambdaout \) 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 \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \) be processes. Following Lemma 3.20, it is sufficient to show that interaction preserves singularity. Let \( u \in \sigma \upharpoonright \tau \). 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 \( e \approx_{u|A+B} e' \) (because projection preserves the temporal order by Lemma 3.19) and \( \lambda_{u|A+B}(e) = \lambda_{u|A+B}(e') \). Since \( u \upharpoonright A + B \in \sigma \) and \( \sigma \) is a process, we deduce that \( e = e' \).

- If \( \lambda_u(e), \lambda_u(e') \in L_C \), then \( e \approx_{u|B+C} e' \) (because projection preserves the temporal order by Lemma 3.19) and \( \lambda_{u|B+C}(e) = \lambda_{u|B+C}(e') \). Since \( u \upharpoonright B + C \in \tau \) and \( \tau \) 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 \( e = e' \).
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 \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \) be processes. We begin by showing that \( \sigma \parallel \tau \) is prefix-closed. Take \( u \cdot u' \in \sigma \parallel \tau \). 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 \( \sigma \) and \( \tau \) 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 \( u \in \sigma \parallel \tau \). So, \( \sigma \parallel \tau \) is prefix-closed.

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

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

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

Let \( u_2 \) be defined as follows.

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

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 \( \sigma \parallel \tau \) is prefix-closed, \( u_1 \in \sigma \parallel \tau \) and hence \( \nu \in \sigma ; \tau \).
Now, we can prove that the basic axioms of a category hold.

**Lemma 3.24.** For each object $A$ in $\text{SynProc}$, $id_A$ is the identity morphism.

**Proof.** Let $\sigma : A \rightarrow B$ be a process and $id_B : B \rightarrow B'$. 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 $u \in \text{int}(A, B, B')$

\[
\text{if } u \upharpoonright A + B = s \in \sigma \quad (3.1)
\]

and $u \upharpoonright B + B' \in id_B \quad (3.2)$

then $u \upharpoonright A + B' = s \quad (3.3)$

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

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

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

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

and is a function. Therefore, $\phi$ is a bijection. Next, we show that $e_1 \preceq_s e_2 \iff \phi(e_1) \preceq_{u \upharpoonright A + B'} \phi(e_2)$.

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 $\phi(e) \preceq_{u \upharpoonright A + B'} \phi(e')$
- if $e \preceq_s e'$ and $\phi(e) = e$ and $\phi(e') \approx_u e'$, then $\phi(e) \preceq_{u \upharpoonright A + B'} \phi(e')$
- if $e \preceq_s e'$ and $\phi(e) \approx_u e$ and $\phi(e') = e'$, then $\phi(e) \preceq_{u \upharpoonright A + B'} \phi(e')$
- if $e \preceq_s e'$ and $\phi(e) \approx_u e$ and $\phi(e') = e'$, then $\phi(e) \preceq_{u \upharpoonright A + B'} \phi(e')$
For the right to left implication, we use the definition of $\phi^{-1}$. For all $e, e' \in E_{u|A+B'}$,

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

Finally, we show that $\lambda_s = \lambda_{u|A+B'} \circ \phi$. First, note that domains of the two functions are $\phi$-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 $\lambda_s(e) = \lambda_{u|A+B'} \circ \phi(e)$.

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

Next, we prove $\sigma \subseteq \sigma; id_B$. Let $s \in \sigma$. We will find an interaction trace $u \in \text{int}(A, B, B')$ 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.

- $E_u = E_s|A + E_s|B + E_s|B$. Let us refer to these subsets using injections $\text{inl}_1 : E_s|A \rightarrow E_u$, $\text{inl}_2 : E_s|B \rightarrow E_u$ and $\text{inl}_3 : E_s|B \rightarrow E_u$.

- $\lambda_u = \lambda_s|A + \lambda_s|B + \lambda_s|B$

- The preorder $\preceq_u$ is defined as follows. For all $e, e' \in E_s$,

  - **P1** if $e, e' \in E_s|A$, then $e \preceq_s e' \iff \text{inl}_1(e) \preceq_u \text{inl}_1(e')$

  - **P2** if $e \in E_s|A$ and $e' \in E_s|B$, then $[e \preceq_s e' \iff \text{inl}_1(e) \preceq_u \text{inl}_2(e') \iff \text{inl}_1(e) \preceq_u \text{inl}_3(e')]$

  - **P3** if $e \in E_s|B$ and $e' \in E_s|A$, then $[e \preceq_s e' \iff \text{inl}_2(e) \preceq_u \text{inl}_1(e') \iff \text{inl}_3(e) \preceq_u \text{inl}_1(e')]$

  - **P4** if $e, e' \in E_s|B$, then $[e \preceq_s e' \iff \text{inl}_2(e) \preceq_u \text{inl}_2(e') \iff \text{inl}_3(e) \preceq_u \text{inl}_3(e') \iff \text{inl}_2(e) \preceq_u \text{inl}_2(e')]$
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 $(\forall e \in E_s|B)(in_2(e) \approx u in_3(e))$. We also have $\lambda_{u|B+B'}(in_2(e)) = inl(b)$ if and only if $\lambda_{u|B+B'}(in_3(e)) = inr(b), b \in L_B$. So, $u \upharpoonright B + B' \in id_B$. We briefly argue that $u \upharpoonright A + B = s$. We have $E_{u|A+B} = E_s|A + E_s|B$. There is a bijection $\phi : E_s \to E_{u|A+B}$ 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 $\lambda_{u|A+B}(e \preceq_s e')$ iff $\lambda_{u|A+B}(\phi(e') \preceq_u in_k(e'))$. We have $e \preceq_s e' \iff in_j(e) \preceq_u in_k(e') \iff in_j(e) \preceq_u in_k(e')$ where $j, k \in \{1, 2\}$.

The proof that $u \upharpoonright A + B' = s$ 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 $s; t = \{ u \in int(A, B, C) \mid u \upharpoonright A + B = s \text{ and } u \upharpoonright B + C = t \}$ and their composition by $s; t = \{ u \upharpoonright A + C \mid u \in s; t \}$. 
Lemma 3.26. Let $s$ be an interaction trace over $\text{int}(A,B,C)$ and $t$ be a trace over $C \rightarrow D$. We have $(s \upharpoonright_B^{A+C}) \downharpoonright t = (s \downharpoonright t) \upharpoonright_B^{A+C+D}$.

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

- $E_\nu = E_{u \upharpoonright A} + E_{s \upharpoonright B} + E_{u \upharpoonright C} + E_{u \upharpoonright D}$. 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.

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

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

$$- (\forall e, e' \in E_{u \upharpoonright A} + E_{s \upharpoonright B} + E_{u \upharpoonright C} + E_{u \upharpoonright D})(e \preceq_1 e' \iff e \preceq_1 e')$$

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

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_\nu)(e' \preceq_1 e''$ and $e'' \preceq_1 e$) if and only if $e' \preceq_2 e$ 

$$- (\exists e'' \in E_\nu)(e \preceq_2 e''$ and $e'' \preceq_2 e$) or $(\forall e'' \in E_\nu)((e \npreceq_1 e''$ or $e'' \npreceq_1 e$) and $(e' \npreceq_1 e''$ or $e'' \npreceq_1 e$)) if and only if $e \preceq_2 e'$

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

We first show that $\preceq_\nu$ is a total preorder. Since $\preceq_u$ and $\preceq_s$ are reflexive, we know that $\preceq_1$ is reflexive. So, $\preceq_\nu$ is reflexive. Totality of $\preceq_\nu$ is ensured by $\preceq_2$. Next, we show that $\preceq_\nu$ is transitive, i.e. for all $e, e', e'' \in E_\nu$, if $e \preceq_\nu e'$ and $e' \preceq_\nu e''$, then $e \preceq_\nu e''$. Let $e \preceq_\nu e'$ and $e' \preceq_\nu e''$. 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 \( e \in E_{u|B} \) and \( [e' \in E_{u|A} \text{ or } e' \in E_{u|C}] \) and \( e'' \in E_{u|D} \), we use the definition of \( \preceq_2 \) to conclude that \( e \preceq_v e'' \).

2. If \( e \in E_{u|A} \) and \( e' \in E_{u|B} \) and and \( e'' \in E_{u|D} \), 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 \( e \preceq_v e' \iff e \preceq_s e' \) and \( e' \preceq_1 e_0 \). Since \( \preceq_s \) is transitive, we get \( e \preceq_s e_0 \). Since \( u \uparrow A + C = s \uparrow A + C \), we get \( e \preceq_u e_0 \). We also have \( e_0 \preceq_1 e'' \iff e_0 \preceq_u e'' \). 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 \((\ast)\). 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 \( \preceq_u \), this is equivalent to \( e'' \preceq_1 e \). Since \( e \preceq_v e' \iff e \preceq_s e' \iff e \preceq_1 e' \), it contradicts \((\ast)\). So, \( e \preceq_u e'' \) and therefore \( e \preceq_v e'' \).

3. If \( e \in E_{u|B} \) and \( e' \in E_{u|D} \) and and \( e'' \in E_{u|B} \), 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 \( u \uparrow A + C = s \uparrow A + C \). 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 \((\dagger)\). 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 \( \preceq_u \), this is equivalent to \( e'' \preceq_1 e \). We also have \( e_2 \preceq_1 e'' \iff e_2 \preceq_s e'' \). Since \( \preceq_s \) is transitive, we get \( e_2 \preceq_s e \iff e_2 \preceq_1 e \). However, we already have \( e' \preceq_1 e_2 \) so \( e' \preceq_1 e_2 \) and \( e_2 \preceq_1 e \) which contradicts \((\dagger)\). So, \( e \preceq_s e'' \) and therefore \( e \preceq_v e'' \).
We need to prove that for all events and $s \leq 3$. CATEGORICAL STRUCTURE

Let $s$ be a trace in $E_{s|A}$. Lemma 3.28. Let $s$ be a trace in $E_{s|A}$. We have $E_{v|A}$ defined as

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

We need to prove that for all $e, e' \in E_s$, we have $e \leq s e'$ iff $\phi(e) \leq_{v|A+B+C} \phi(e')$. We have $e \leq s e' \iff in_j(e) \leq_1 in_k(e') \iff in_j(e) \leq v in_k(e') \iff in_j(e) \leq_{v|A+B+C} in_k(e')$ where $j, k \in \{1, 2, 3\}$.

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

$$
\psi(e) = \begin{cases} 
    in_1(e) & \text{if } \lambda_u(e) \in \text{inl}(L_A) \\
    in_2(e) & \text{if } \lambda_u(e) \in \text{inr}(L_C) \\
    in_3(e) & \text{if } \lambda_u(e) \in \text{inr}(L_D) 
\end{cases}
$$

We need to prove that for all $e, e' \in E_u$, we have $e \leq u e'$ iff $\psi(e) \leq_{v|A+C+D} \psi(e')$. We have $e \leq u e' \iff in_j(e) \leq_1 in_k(e') \iff in_j(e) \leq v in_k(e') \iff in_j(e) \leq_{v|A+C+D} in_k(e')$ where $j, k \in \{1, 2, 3\}$.

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

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

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

**Lemma 3.28.** Let $s$ be a trace in $\Theta(A + B + C)$. We have $(s \uparrow_{C}^{A+B}) \uparrow_{B}^{A} = s \uparrow_{B+C}^{A}$. 
**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, $(\sigma\lhd\tau)\lhd\gamma = \sigma\lhd(\tau\lhd\gamma)$. We expand $(\sigma\lhd\tau)\lhd\gamma$ using Definition 3.12,

\[
(\sigma\lhd\tau)\lhd\gamma = \{u \in \text{int}(A,B,C,D) \mid u \upharpoonright_{A+B+C+D} C \subseteq \sigma \quad \text{and} \quad u \upharpoonright_{A+B} D \subseteq \tau \quad \text{and} \quad u \upharpoonright_{A+C} B \subseteq \gamma \}
\]

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

\[
\sigma\lhd(\tau\lhd\gamma) = \{u \in \text{int}(A,B,C,D) \mid u \upharpoonright_{A+B+C+D} C \subseteq \sigma \quad \text{and} \quad u \upharpoonright_{A+B} D \subseteq \tau \quad \text{and} \quad u \upharpoonright_{A+C} B \subseteq \gamma \}
\]

Using Lemma 3.28, we get,

\[
(\sigma\lhd\tau)\lhd\gamma = \sigma\lhd(\tau\lhd\gamma)
\]

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

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 $\sigma; \tau = \sigma; \tau \upharpoonright_{A^+C}$. Hence,

$$
(\sigma; \tau)\; \gamma = ((\sigma; \tau \upharpoonright_{A^+C}) \; \gamma) \upharpoonright_{A^+D} \\
= ((\sigma; \tau \gamma) \upharpoonright_{B^+C}) \upharpoonright_{A^+D} \text{ by Corollary 3.27} \\
= (\sigma; \gamma) \upharpoonright_{B^+C} \text{ by Lemma 3.28} \\
= (\sigma; \tau \gamma) \upharpoonright_{B^+C} \text{ by (★)} \\
= (\sigma; (\tau \gamma \mid_{B^+D})) \upharpoonright_{B^+D} \text{ by Lemma 3.28} \\
= (\sigma; (\tau \gamma)) \upharpoonright_{B^+D} \text{ by Corollary 3.27} \\
= \sigma; (\tau; \gamma).
$$

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 $\otimes : \text{SynProc} \times \text{SynProc} \rightarrow \text{SynProc}$ defined as follows.

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

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

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

**Lemma 3.32.** The tensor product preserves singularity.
Proof. Let $\sigma : A \rightarrow B$ and $\tau : C \rightarrow D$ 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 $e \equiv_u e'$ when $e \approx_u e'$ and $[\lambda_u(e) = inl(a)$ if and only if $\lambda_u(e') = inl(a)]$ 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 $u \upharpoonright A + A' \in id_A$ and $u \upharpoonright B + B' \in id_B$. Using Definition 3.15, we get $u \in \Theta(A \otimes B \Rightarrow A' \otimes B')$ and $(\forall e \in E_u \upharpoonright A + A', \exists e' \in E_u \upharpoonright A + A')(e \equiv_{u \upharpoonright A + A'} e')$ and $(\forall e \in E_u \upharpoonright B + B', \exists e' \in E_u \upharpoonright B + B')(e \equiv_{u \upharpoonright B + B'} e')$. Since $E_u = E_u \upharpoonright A + A' \cup E_u \upharpoonright B + B'$ and $(\forall e, e' \in E_u)(e \approx_{u \upharpoonright A + A'} e' \lor e \approx_{u \upharpoonright B + B'} e'$), we conclude that for all $e \in E_u$, there is $e' \in E_u$ such that $e \equiv_u e'$. 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 $(\forall e \in E_u, \exists e \in E_u)(e \equiv_u e')$. After projecting $u$ over $A + A'$, we get $u \upharpoonright A + A' \in \Theta(A \Rightarrow A')$ and $(\forall e \in E_u \upharpoonright A + A', \exists e' \in E_u \upharpoonright A + A')(e \equiv_{u \upharpoonright A + A'} e')$; that is, $u \upharpoonright A + A' \in id_A$. Projecting $u$ over $B + B'$ yields $u \upharpoonright B + B' \in \Theta(B \Rightarrow B')$ and $(\forall e \in E_u \upharpoonright B + B', \exists e' \in E_u \upharpoonright B + B')(e \equiv_{u \upharpoonright B + B'} e')$; that is, $u \upharpoonright B + B' \in id_B$. So, $u \in id_A \otimes id_B$. □

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

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

**Lemma 3.36.** The tensor product preserves composition.

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

$LHS = \{u \in \Delta(A \otimes D \Rightarrow B \otimes E) \Rightarrow C \otimes F) \mid u \upharpoonright A + B + C \in \sigma \upharpoonright_B \tau$ and $u \upharpoonright D + E + F \in \alpha \upharpoonright_B \beta\}$
3.2. CATEGORICAL STRUCTURE

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

\[ \text{RHS} = \{ u \in \Delta ((A \otimes D \Rightarrow B \otimes E) \Rightarrow C \otimes F) \mid u \upharpoonright A+B+D+E \in \sigma \otimes \alpha \text{ and } \]
\[ 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 (*) Next, we show that \((\sigma; \tau) \otimes (\alpha; \beta) = (\sigma \otimes \alpha); (\tau \otimes \beta)\).

\[(\sigma; \tau) \otimes (\alpha; \beta) = (\sigma \upharpoonright (A+C) \otimes (\alpha \upharpoonright (D+F)) \text{ by Definition 3.13}
\[ = ((\sigma \upharpoonright \tau) \otimes (\alpha \upharpoonright \beta)) \upharpoonright A+C+D+F \text{ by Lemma 3.35}
\[ = ((\sigma \otimes \alpha) \upharpoonright (\tau \otimes \beta)) \upharpoonright A+C+D+F \text{ by (*)}
\[ = (\sigma \otimes \alpha); (\tau \otimes \beta) \]

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,

\[ \alpha_{A,B,C} : (A \otimes B) \otimes C \rightarrow A \otimes (B \otimes C) \]

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 \((A \otimes B) \otimes C \rightarrow A \otimes (B \otimes C)\) 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 \textit{inl} and \textit{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 \( \alpha \).

**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,

\[
\lambda_A : I \otimes A \rightarrow A \\
\rho_A : A \otimes I \rightarrow A
\]

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 \( A \in \text{SynProc} \), \( I \otimes A \) 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 \( \lambda_A \) and the right identity isomorphism \( \rho_A \) as those with the signatures above, and which consist of exactly the same traces, up to retagging of labels by \( \text{inl} \) and \( \text{inr} \), as \( \text{id}_A \). The coherence condition in Figure 3.2 directly follows from the definitions of the tensor product, \( \lambda \), and \( \rho \).

In the next part, we show that \( \text{SynProc} \) is symmetric.

**Proposition 3.39.** \( \text{SynProc} \) 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, \( \gamma_{A,B} : A \otimes B \rightarrow B \otimes A \) such that the hexagon diagrams in Figure 3.3 commute and \( \gamma_{A,B} ; \gamma_{B,A} = \text{id}_{A \otimes B} \).

**Proof.** For any \( A, B \in \text{SynProc} \), let \( \gamma_{A,B} : A \otimes B \rightarrow B \otimes A \) be the isomorphism consisting of exactly the same traces, up to retagging of labels by \( \text{inl} \) and \( \text{inr} \), as \( \text{id}_{A \otimes B} \). 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 \( \text{inl} \) and \( \text{inr} \), as \( \text{id}_{A \otimes B} ; \text{id}_{A \otimes B} \). However, as \( \gamma_{A,B} \) changes the tags of labels and \( \gamma_{B,A} \) changes them back according to (*) \( \gamma_{A,B} ; \gamma_{B,A} = \text{id}_{A \otimes B} ; \text{id}_{A \otimes B} = \text{id}_{A \otimes B} \).

\[
\begin{align*}
L_{A \otimes B \Rightarrow A' \otimes B'} &= \text{inl}(\text{inl}(A)) \cup \text{inl}(\text{inr}(B)) \cup \text{inr}(\text{inl}(A')) \cup \text{inr}(\text{inr}(B')) \\
L_{A \otimes B \Rightarrow B' \otimes A'} &= \text{inl}(\text{inl}(A)) \cup \text{inl}(\text{inr}(B)) \cup \text{inr}(\text{inl}(B')) \cup \text{inr}(\text{inl}(A')) \\
L_{B' \otimes A' \Rightarrow A \otimes B} &= \text{inl}(\text{inl}(B')) \cup \text{inl}(\text{inr}(A')) \cup \text{inr}(\text{inl}(A)) \cup \text{inr}(\text{inr}(B)) \tag{*}
\end{align*}
\]
3.2. CATEGORICAL STRUCTURE

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

We can now state the following.

**Proposition 3.40.** $\text{SynProc}$ is a symmetric monoidal category.
A \otimes (B \otimes C) \xrightarrow{\alpha_{A,B,C}^{-1}} (A \otimes B) \otimes C \xrightarrow{\gamma_{A,B} \otimes id_C} (B \otimes A) \otimes C

\begin{align*}
\gamma_{A,B} \otimes C &\xrightarrow{id_B \otimes \gamma_{A,C}} B \otimes (C \otimes A) \\
B \otimes (C \otimes A) &\xrightarrow{\alpha_{B,C,A}^{-1}} (B \otimes C) \otimes A
\end{align*}

\begin{align*}
(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) \\
\gamma_{A \otimes B,C} &\xrightarrow{\alpha_{A,C,B}^{-1}} (C \otimes (A \otimes B)) \xrightarrow{\alpha_{C,A,B}} (C \otimes A) \otimes B \xrightarrow{\gamma_{A,C} \otimes id_B} (A \otimes C) \otimes B
\end{align*}

Figure 3.3: Coherence conditions for $\gamma$

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

\[
\lambda_{\varphi(s)}(e) = \begin{cases}
  \text{inl}(a) & \text{if } \lambda_e(e) = \text{inl}(\text{inl}(a)), a \in L_A \\
  \text{inr}(\text{inl}(b)) & \text{if } \lambda_e(e) = \text{inl}(\text{inr}(b)), b \in L_B \\
  \text{inr}(\text{inr}(c)) & \text{if } \lambda_e(e) = \text{inr}(c), c \in L_C 
\end{cases}
\]

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

We can now show that SynProc is closed.

**Proposition 3.42.** SynProc with $- \otimes -$, I, $\text{eval}_{A,B}$ 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,
3.2. CATEGORICAL STRUCTURE

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

\[
\text{if } u \mid A + B + C = t \in \Lambda_{A,B,C}(\sigma) \\
\quad \text{and } u \mid B' + B'' \in id_B \\
\quad \text{and } u \mid B + B'' \in id_B \\
\quad \text{and } u \mid C + C' \in id_C \\
\text{then } u \mid A + B' + C' = \psi^{-1}(t)
\]

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 \( \alpha : E_s \rightarrow E_u|A+B'+C' \) such that \( (\forall e_1, e_2 \in E_s)(e_1 \preceq_s e_2 \iff \alpha(e_1) \preceq_u \alpha(e_2)) \) and \( \lambda_s = \lambda_{u|A+B'+C'} \circ \alpha \). We first define a bijection \( \phi : E_t \rightarrow E_u|A+B'+C' \) as follows. For all \( e \in E_t \),

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

We know \( \phi \) is well-defined because the definition of \( id \) ensures the existence and singularity ensures the uniqueness of the output. In addition, its inverse \( \phi^{-1} \), defined as follows for all \( e \in E_u|A+B'+C' \), is a function.

- If \( \lambda_{u|A+B'+C'}(e) \in \text{inl}(\text{inl}(L_A)) \), then \( \phi^{-1}(e) = e \).
- If \( \lambda_{u|A+B'+C'}(e) \in \text{inr}(\text{inr}(L_B)) \), then there is \( e'' \in E_u \) such that \( \phi^{-1}(e) = e_1 \) and \( e_1 \approx_u e'' \approx_u e \) and \( \lambda_u(e_1) = \text{inl}(\text{inr}(\text{inl}(\text{inl}(b)))) \) and \( \lambda_u(e'') = \text{inl}(\text{inr}(\text{inr}(b))) \) and \( \lambda_u(e) = \text{inl}(\text{inr}(\text{inr}(b))) \) and \( b \in L_B \).
- If \( \lambda_{u|A+B'+C'}(e) \in \text{inr}(L_C) \), then \( \phi^{-1}(e) = e_2 \) where \( e_2 \approx_u e \) and \( \lambda_u(e_2) = \text{inl}(\text{inr}(\text{inl}(c))) \) and \( \lambda_u(e) = \text{inr}(c) \), \( c \in L_C \).
Therefore, \( \phi \) is a bijection. Since \( E_s = E_t \), it follows \( \alpha = \phi \circ id \) is a bijection. We can show that
\[
e_1 \preceq_t e_2 \text{ iff } \phi(e_1) \preceq_{u|A+B+C'} \phi(e_2)
\]
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 \( e_1 \preceq_s e_2 \text{ iff } \alpha(e_1) \preceq_{u|A+B+C'} \alpha(e_2) \).

Next, we show that \( \lambda_s = \lambda_{u|A+B'+C'} \circ \alpha \). 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 \( \lambda_s(e) = \lambda_{u|A+B'+C'} \circ \alpha(e) \).

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

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

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

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

- \( E_u = E_s|B + E_s|A + E_s|B + E_s|B + E_s|C + E_s|C \). Let us refer to these subsets using injections \( in_1 \) to \( in_6 \).
- \( E_u = \lambda_s|B + \lambda_s|A + \lambda_s|B + \lambda_s|B + \lambda_s|C + \lambda_s|C \).
- The preorder \( \preceq_u \) is defined as follows. For all \( e, e' \in E_s \),
  
  \( P1 \) if \( e, e' \in E_s|A \), then \( e \preceq_s e' \iff in_2(e) \preceq_u in_2(e') \)
  
  \( P2 \) if \( e, e' \in E_s|B \), then \( e \preceq_s e' \iff in_1(e) \preceq_u in_1(e') \iff in_1(e) \preceq_u in_3(e') \iff in_1(e) \preceq_u in_4(e') \iff in_1(e') \iff in_4(e) \preceq_u in_4(e') \),
  
  \( P3 \) if \( e, e' \in E_s|C \), then \( e \preceq_s e' \iff in_5(e) \preceq_u in_5(e') \) and \( in_6(e) \preceq_u in_6(e') \) and \( in_5(e) \preceq_u in_6(e') \) and \( in_6(e') \) and \( in_5(e) \preceq_u in_5(e') \)
  
  \( P4 \) if \( e \in E_s|A \) and \( e' \in E_s|B \), then \( e \preceq_s e' \iff in_2(e) \preceq_u in_2(e') \iff in_2(e) \preceq_u in_3(e') \iff in_2(e) \preceq_u in_4(e') \).
3.2. CATEGORICAL STRUCTURE

\( P_5 \) if \( e \in E_{s|A} \) and \( e' \in E_{s|C} \), then \([e \preceq_s e' \iff \text{in}_2(e) \preceq_u \text{in}_5(e') \iff \text{in}_2(e) \preceq_u \text{in}_6(e')]\)

\( P_6 \) if \( e \in E_{s|B} \) and \( e' \in E_{s|A} \), then \([e \preceq_s e' \iff \text{in}_1(e) \preceq_u \text{in}_2(e') \iff \text{in}_3(e) \preceq_u \text{in}_2(e') \iff \text{in}_4(e) \preceq_u \text{in}_2(e')]\)

\( P_7 \) if \( e \in E_{s|B} \) and \( e' \in E_{s|C} \), then \([e \preceq_s e' \iff \text{in}_3(e) \preceq_u \text{in}_5(e') \iff \text{in}_3(e) \preceq_u \text{in}_6(e') \iff \text{in}_4(e) \preceq_u \text{in}_5(e') \iff \text{in}_4(e) \preceq_u \text{in}_6(e')]\)

\( P_8 \) if \( e \in E_{s|C} \) and \( e' \in E_{s|A} \), then \([e \preceq_s e' \iff \text{in}_5(e) \preceq_u \text{in}_2(e') \iff \text{in}_6(e) \preceq_u \text{in}_2(e')]\)

\( P_9 \) if \( e \in E_{s|C} \) and \( e' \in E_{s|B} \), then \([e \preceq_s e' \iff \text{in}_5(e) \preceq_u \text{in}_6(e') \iff \text{in}_4(e) \preceq_u \text{in}_5(e') \iff \text{in}_6(e) \preceq_u \text{in}_4(e')]\)

We use the same strategy as the proof of Lemma 3.24 to show that \( \preceq_u \) is a total preorder and \( u \) respects singularity. We can also check, following the structure of the proof of Lemma 3.24 that \( u \) respects the conditions set out above. \( \blacksquare \)
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]*
CHAPTER 4. COMPOSITIONAL ROUND ABSTRACTION

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
is achieved using the operator $\text{next}$: given a module $P$ and a set $Y$ of interface variables, the module $\text{next } Y$ 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 $\text{next } eq$ for $\text{Counter}$ is depicted in Figure 4.2.

One can think of round abstraction in slightly broader terms [AHR98]: if $M$ is a system specification, and $\phi$ is a condition on the variables of $M$, then the round abstraction $\text{next } \phi$ for $M$ gathers as many transitions of $M$ as necessary to satisfy $\phi$, 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 $\text{RACounter}$
  external $\text{tick : } \mathbb{B}$
  interface $\text{eq : } \mathbb{B}$
  private $\text{roundcount, tickcount : } \mathbb{N}$
  atom $\text{eq reads } \text{roundcount, tickcount}$
  init update
    [] $\text{roundcount = tickcount * 2} \rightarrow \text{eq!}$

Figure 4.2: A round abstracted $\text{Counter}$
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 $\sigma : A \rightarrow B$ and $\tau : B \rightarrow C$, that if $\sigma'$ is a round abstraction of $\sigma$ and $\tau'$ is a round abstraction of $\tau$, 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.
4.2. ROUND ABSTRACTION ON PROCESSES

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 \( s = (E_s, \preceq_s, \lambda_s) \), \( t = (E_t, \preceq_t, \lambda_t) \) 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 \( (E_s, \lambda_s) \) and \( (E_t, \lambda_t) \) are \( \phi \)-isomorphic, i.e. \( \lambda_s = \lambda_t \circ \phi \), and \( \phi \) 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 \( \tau \), the ‘abstracted’ process, has no ‘junk’ traces which do not stem from \( \sigma \). A stronger property, total round abstraction, written \( \sigma \sqsubseteq \sim \tau \), additionally requires that all the behaviour of \( \sigma \) can be found, in an abstracted form, in \( \tau \). In general, it is not the case that \( \sigma \sqsubseteq \sigma' \) and \( \tau \sqsubseteq \tau' \) implies \( \sigma; \tau \sqsubseteq \sigma'; \tau' \), or \( \sigma \sqsubseteq \sigma' \) and \( \tau \sqsubseteq \tau' \) implies \( \sigma; \tau \sqsubseteq \sigma'; \tau' \). 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.

\[
\sigma = pc(\{b_2, b_1.a\}) \sqsubseteq \sigma' = pc(\{(b_2, b_1).a\})
\]
\[
\tau = pc(\{c.b_1.b_2\}) \sqsubseteq \tau' = pc(\{c.(b_1, b_2)\})
\]

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

**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.

\[
\sigma = pc(\{b_1, b_2, a\}) \quad \subseteq \quad \sigma' = pc(\{\langle b_1, b_2 \rangle, a\}) \\
\tau = pc(\{c, b_1, b_2\}) \quad \subseteq \quad \tau' = pc(\{c, b_1, b_2\})
\]

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

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 $\sigma$ does not coincide with the order in which they can be received by $\tau$; 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 $\sigma'$ and consecutive in $\tau'$ 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 $\sigma$ and $\tau$ over $A$, we say that $\tau$ is a partial round abstraction of $\sigma$, written $\sigma \sqsubseteq \tau$, if for any $t \in \tau$ there is $s \in \sigma$ such that $s \sqsubseteq t$.

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 $s \sqsubseteq s'$, then $s \upharpoonright A \sqsubseteq s' \upharpoonright A$.

**Proof.** Projection does not affect the temporal order of those events that remain after projection.

By definition, $s \sqsubseteq s'$ 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 $\preceq_{s \upharpoonright A} \subseteq \preceq_{s' \upharpoonright A}$. $\blacksquare$
4.3. COMPOSITIONALITY OF PARTIAL ROUND ABSTRACTION

Corollary 4.6. For processes \( \sigma, \sigma' : A \rightarrow B \) and \( \tau, \tau' : B \rightarrow C \), if \( \sigma \vdash \tau \supseteq \sigma' \vdash \tau' \), then \( \sigma; \tau \supseteq \sigma'; \tau' \).

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 \( \phi : E_s \rightarrow E_t \) 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 \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \) are compatible, written \( \sigma \times \tau \), when the following conditions hold.

1. For any \( s \in \sigma \), and any \( u \in \text{int}(A, B, C) \) such that \( u \upharpoonright B + C \in \tau \), if
   
   \[ \begin{align*} 
   & \bullet \ u \upharpoonright B \in \Pi(s \upharpoonright B) \\
   & \bullet \text{ and } u \upharpoonright A = s \upharpoonright A 
   \end{align*} \]
   
   then \( u \upharpoonright A + B \in \sigma \).

2. For any \( t \in \tau \), and any \( u \in \text{int}(A, B, C) \) such that \( u \upharpoonright A + B \in \sigma \), if
   
   \[ \begin{align*} 
   & \bullet \ u \upharpoonright B \in \Pi(t \upharpoonright B) \\
   & \bullet \text{ and } u \upharpoonright C = t \upharpoonright C 
   \end{align*} \]
   
   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.
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 $\sigma' : A \rightarrow B$ and $\tau' : B \rightarrow C$ are post-compatible with respect to processes $\sigma : A \rightarrow B$ and $\tau : B \rightarrow C$, respectively, written $\sigma' \circ \tau'$, when the following conditions hold.

1. For any $s \in \sigma$ and any $u \in \text{int}(A, B, C)$ 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 $u \in \text{int}(A, B, C)$ 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 \( \sigma \subseteq \sigma' \) and \( \tau \subseteq \tau' \). For all \( u' \in \sigma' \downarrow \tau' \), there are interaction traces \( u, v \in \text{int}(A, B, C) \) and traces \( s \in \sigma \) and \( t \in \tau \) satisfying,

1. \( u \subseteq u' \) and \( u \upharpoonright A + B = s \) and \( u \upharpoonright B = t \)
2. \( v \subseteq u' \) and \( v \upharpoonright B + C = t \)

Proof. Let \( u' \in \sigma' \downarrow \tau' \); that is, \( u' \upharpoonright A = \sigma' \) and \( u' \upharpoonright B = \tau' \). Since \( \sigma \subseteq \sigma' \) and \( \tau \subseteq \tau' \), it follows there are \( s \in \sigma \) and \( t \in \tau \) such that \( s \subseteq u' \upharpoonright A + B \) and \( t \subseteq u' \upharpoonright B + C \). So, let \( \phi : E_s \to E_{u'|A+B} \) and \( \psi : E_t \to E_{u'|B+C} \) be bijections such that \( \lambda_s = \lambda_{u'|A+B} \circ \phi \) and \( \lambda_t = \lambda_{u'|B+C} \circ \psi \). Without loss of generality, we will assume that \( \phi \) and \( \psi \) are identities, implying that \( E_s = E_{u'|A+B} \) and \( E_t = E_{u'|B+C} \). It follows that \( E_s|B = E_t|B \) 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 \( < \), defined as \( a < b \) 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 \( e, e' \in E_u \).

  \[
  \begin{align*}
  P_1 & \quad e \prec_{u'} e' \iff e \preceq_{1} e'. \\
  P_2 & \quad \text{If } e, e' \in E_s, \text{ then } e \prec_s e' \iff e \preceq_{2} e'. \\
  P_3 & \quad \text{If } e, e' \in E_t|C, \text{ then } e \preceq_t e' \iff e \preceq_{2} e'. \\
  P_4 & \quad \text{If } e \in E_t|C \text{ and } e' \in E_s|B, \text{ then } e \approx_{u'} e' \iff e \preceq_{2} e'. \\
  P_5 & \quad \text{If } e \in E_t|C \text{ and } e' \in E_s|A, \text{ then } e \approx_{u'} e' \iff e \preceq_{2} e'. \\
  P_6 & \quad \text{If } e \in E_s \text{ and } e' \in E_t|C, \text{ then } e \preceq_{2} e'.
  \end{align*}
\]

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. \( (\forall e, e' \in E_u)(e \preceq_u e' \text{ or } e' \preceq_u e) \)
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 \( [\lambda_u(e) \in L_A \text{ or } \lambda_u(e) \in L_B] \) and \( [\lambda_u(e') \in L_A \text{ or } \lambda_u(e') \in L_B] \), then \( e \preceq s e' \text{ or } e' \preceq s e \) since \( \preceq s \) is total. Using (P2), we get \( e \preceq u e' \text{ or } e' \preceq u e \).

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 \( \preceq u \) is total.

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

4. If \( [\lambda_u(e) \in L_C \text{ and } \lambda_u(e') \in L_A] \) or \( [\lambda_u(e) \in L_A \text{ and } \lambda_u(e') \in L_C] \), then given that \( \preceq u' \) is total, we have \( e \preceq u' e' \text{ or } e' \preceq u' e \). We can rewrite this as \( e \prec u' e' \text{ or } e' \prec u' e \text{ or } e \approx u' e' \). 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 \( e \preceq_{u'} e' \text{ and } e' \not\preceq_{u'} e \text{ and } e' \preceq_{u'} e'' \). Since \( \preceq_{u'} \) is transitive, we get \( e \approx_{u'} e'' \). Suppose to the contrary that \( e'' \not\preceq_{u'} e \). Then, as \( \preceq_{u'} \) is transitive, we get \( e' \not\preceq_{u'} e \) 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.
4.3. COMPOSITIONALITY OF PARTIAL ROUND ABSTRACTION

- Case \((C, C, C)\). We use (P3) and the fact that \(\preceq_t\) is transitive.
- Cases \((A, A, C), (A, B, C), (A, C, A), (A, C, B), (B, A, C), (B, B, C), (B, C, A), (B, C, C), (C, A, C), (C, B, C)\) 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 \(e' \preceq_s e''\). 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 \(E_{u|B} = E_{u'|B} = E_{t|B}\) and \(\lambda_{u|B} = \lambda_{u'|B} = \lambda_{t|B}\).
- Next, we need to show that \(u \upharpoonright A + B = s\). We have \(E_{u|A+B} = E_{u'|A+B} = E_s\) and \(\lambda_{u|A+B} = \lambda_{u'|A+B} = \lambda_s\). We need to show that \(\preceq_s = \preceq_{u|A+B}\). The left to right inclusion follows from (P2). For the right to left inclusion, take \(e, e' \in E_{u|A+B}\) such that \(e \preceq_{u} e'\). Using the definition of \(\preceq_s\), 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 \(u \upharpoonright A + B = s\).
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. $e \prec_{u'} e' \iff e \preceq_3 e'$.
  2. If $e \in E_t$ and $e' \in E_t$, then $e \preceq_t e' \iff e \preceq_4 e'$.
  3. If $e, e' \in E_{s|A}$, then $e \preceq_s e' \iff e \preceq_4 e'$.
  4. If $e \in E_{s|B}$ and $e' \in E_{s|A}$, then $e \approx_{u'} e' \iff e \preceq_2 e'$.
  5. If $e \in E_{s|C}$ and $e' \in E_{s|A}$, then $e \approx_{u'} e' \iff e \preceq_2 e'$.

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

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

**Proof.** By Corollary 4.6, it is sufficient to show that $\sigma \ll \tau \sqsubseteq \sigma' \ll \tau'$. Let $u' \in \sigma' \ll \tau'$. 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 $u_1 \upharpoonright A + B \in \sigma$ and $u_1 \upharpoonright B \in \Pi(t \upharpoonright B)$ and $u_1 \upharpoonright C = t \upharpoonright C$. Using compatibility, we deduce that $u_1 \upharpoonright B + C$ and hence $u \in \sigma \ll \tau$. Since $u_1 \sqsubseteq u'$ we are done. Note that the proof can also be done symmetrically using $\sigma$ instead of $\tau$.

**Theorem 4.12** (Soundness II). For any processes $\sigma, \sigma' : A \rightarrow B$ and $\tau, \tau' : B \rightarrow C$, if $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\sigma' \circ \tau', then \sigma ; \tau \sqsubseteq \sigma' ; \tau'$. 
4.3. COMPOSITIONALITY OF PARTIAL ROUND ABSTRACTION

Proof. By Corollary 4.6, it is sufficient to show that \( \sigma \perp \tau \subseteq \sigma' \perp \tau' \). Let \( u' \in \sigma' \perp \tau' \). 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

- \( u_1 \subseteq u' \) and \( u_1 \upharpoonright A + B = s \) and \( u_1 \upharpoonright B \in \Pi(t \upharpoonright B) \) and \( u_1 \upharpoonright C = t \upharpoonright C \)
- \( u_2 \subseteq u' \) 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 \( u_1 \subseteq u' \) and \( u_1 \upharpoonright A + B = s \), it follows, by Lemma 4.5, that \( s \subseteq u' \upharpoonright A + B \). Since \( u_2 \subseteq u' \) and \( u_2 \upharpoonright B + C = t \), it follows, by Lemma 4.5, that \( t \subseteq u' \upharpoonright B + C \). Using post-compatibility, we get \( u_1 \upharpoonright B + C \in \tau \) and \( u_2 \upharpoonright A + B \in \sigma \). So, \( u_1, u_2 \in \sigma \perp \tau \). Since \( u_1 \subseteq u' \) and \( u_2 \subseteq u' \), 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 \( \sigma \subseteq \sigma' \) and \( \tau \subseteq \tau' \), then \( \sigma \otimes \tau \subseteq \sigma' \otimes \tau' \).

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

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 \((\forall e, e' \in E_s)(if e \preceq_s e', then \phi(e) \preceq_{v' \upharpoonright A + B} \phi(e') \) and \((\forall e, e' \in E_t)(if e \preceq_t e', then \psi(e) \preceq_{v' \upharpoonright C + D} \psi(e').)

We show that there is a trace \( v \in \sigma \otimes \tau \) such that \( v \subseteq v' \). 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 \( \alpha(e') \npreceq_{v'} \alpha(e) \), 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 \( e \preceq_t e' \iff e \preceq_v e' \)

(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 \( v \upharpoonright A + B = s \) and \( v \upharpoonright B + C = t \). Let \( e, e' \in E_v \upharpoonright A + B = E_s \). We show that \( e \preceq_{v|A+B} e' \) iff \( e \preceq_s e' \). The right to left implication is straightforward. For the left to right implication, let \( e \preceq_{v|A+B} e' \). So, \( e \preceq_v e' \). Since \( v \subseteq v' \), it follows \( \alpha(e) \preceq_{v'} \alpha(e') \). This means either

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

The proof that \( v \upharpoonright B + C = t \) is symmetric. \( \blacksquare \)

### 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 \( \sigma : A \rightarrow B \) and process \( \sigma' : A \rightarrow B \), we say that \( \sigma' \) is a total round abstraction of \( \sigma \), written \( \sigma \sqsubseteq \sigma' \), if \( \sigma \subseteq \sigma' \) 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 \( s \cdot w \subseteq s' \).

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 \( pc(a.b.c) = \{e, a, a.b, a.b.c\} \) whereas \( pc(\langle a, b, c \rangle) = \{e, \langle a, b, c \rangle\} \); using Definition 4.14, we have \( pc(a.b.c) \sqsubseteq pc(\langle a, b, c \rangle) \). Note that, unlike partial round abstraction, the above definition is not transitive: from \( \sigma \sqsubseteq \sigma' \) and \( \sigma' \sqsubseteq \sigma'' \), we cannot conclude that \( \sigma \sqsubseteq \sigma'' \). 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 \( \sigma, \sigma' : A \rightarrow B \) and \( \tau, \tau' : B \rightarrow C \), with \( L_A = \{a\}, L_B = \{b_1, b_2, b_3\}, L_C = \{c\} \), be the following processes.

\[
\begin{align*}
\sigma &= pc(\{b_1.b_2.a\}) \\
\tau &= pc(\{c,b_1,b_3\}) \\
\sigma' &= pc(\{b_1.b_2.a\}) \\
\tau' &= pc(\{c,b_1,b_3\})
\end{align*}
\]

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

In this example, the original processes \( \sigma \) and \( \tau \) compose well up to \( b_1 \) then deadlock as they attempt to synchronise on mismatched events. Because \( \sigma' \) and \( \tau' \) 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 GSSL94 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 $wf.ok.rf.wt.ok.rt$ could be presumably abstracted as $⟨wf, ok, r, f⟩ · ⟨wt, ok, r, t⟩$. 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 $r.f.wf.ok.rt.wt.ok$ can also be abstracted to $⟨wf, ok, r, f⟩ · ⟨wt, ok, r, t⟩$, which is the same as the abstraction of the well-formed trace above. At the level of the programming language, it means that programs $x := 0; x := 1; \text{if } x = 1 \text{ diverge}$ and $x := 0; \text{if } x = 1 \text{ diverge}; x := 1$ 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 $x := !x$ takes time to run. Further, let us take the round abstraction of the no-op command skip as $⟨r, d⟩$. 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 $x := !x; \text{skip}$, the circuit corresponding to sequential composition requires the ability to deal with both commands that take time and those that instantly terminate.
Causality is central to many theoretical frameworks including concurrency theory [Win87, Maz95], synchronous languages [Ber99] and component-based systems [GMRI10].

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 $(L_A, \pi_A, \vdash_A)$ where,

- $L_A$ is a finite set of labels.
- $\pi_A : L_A \to \{i, o\}$ maps each label to an input/output polarity.
- $\vdash_A \subseteq (L_A + \{\ast\}) \times L_A$ is a relation called causality that satisfies the following conditions.

\[-\text{Let } \vdash_A^T \text{ be the transitive closure of } \vdash_A. \text{ For all } a, b \in L_A, \text{ if } a \vdash_A^T b, \text{ then } b \not\vdash_A^T a.\]

\[-\text{If } \ast \vdash_A a, \text{ then } \pi_A(a) = i \text{ and } [b \vdash_A a \iff b = \ast].\]

\[-\text{If } a \vdash_A b \text{ and } a \neq \ast, \text{ then } \pi_A(a) \neq \pi_A(b).\]

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 $\ast$ 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.

\[
\begin{align*}
L_{A \otimes B} &= L_A + L_B \\
\pi_{A \otimes B} &= [\pi_A, \pi_B] \\
\ast \vdash_{A \otimes B} a &\iff \ast \vdash_A a \text{ or } \ast \vdash_B a \\
a \vdash_{A \otimes B} b &\iff a \vdash_A b \text{ or } a \vdash_B b
\end{align*}
\]

\[
\begin{align*}
L_{A \Rightarrow B} &= L_A + L_B \\
\pi_{A \Rightarrow B} &= [\pi_A^*, \pi_B] \\
\ast \vdash_{A \Rightarrow B} a &\iff \ast \vdash_B a \\
a \vdash_{A \Rightarrow B} b &\iff a \vdash_A b \text{ or } a \vdash_B b \text{ or } (\ast \vdash_B a \text{ and } \ast \vdash_A b)
\end{align*}
\]
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 $\langle E, \preceq, \lambda, \circlearrowright \rangle$ where $E$ is a finite set of events, $\preceq$ is a total preorder on $E$, $\lambda : E \rightarrow L_A$ is a function mapping events to labels in $A$ and $\circlearrowright : E \setminus \{e \in E \mid \lambda(e) \in I_A\} \rightarrow E$ is a partial function such that $\circlearrowright(e) = e'$ implies $e' \preceq e$ and $\lambda(e') \vdash_A \lambda(e)$.

If $\circlearrowright(e) = e'$, we write $e' \circlearrowright e$ 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 $\langle E, \preceq, \lambda, \circlearrowright \rangle$ with singular events and where $\circlearrowright$ 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 $s = \langle E_s, \preceq_s, \lambda_s, \circlearrowright_s \rangle$ and $t = \langle E_t, \preceq_t, \lambda_t, \circlearrowright_t \rangle$ are equivalent if there exists a bijection $\phi : E_s \rightarrow E_t$ satisfying for all events $e_1, e_2 \in E_s$

- $e_1 \preceq_s e_2 \iff \phi(e_1) \preceq_t \phi(e_2)$ and
- $\lambda_s = \lambda_t \circ \phi$ and
- $e_1 \circlearrowright_s e_2 \iff \phi(e_1) \circlearrowright_t \phi(e_2)$.

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 \( \lceil - \rceil : \Theta_J(A) \to \Theta((L_A, \pi_A)) \) which maps each justified trace \( u \) to \( (E_u, \preceq_u, \lambda_u) \); that is, \( u \) 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 \( \vartriangleleft_u = \vartriangleleft_u \cap (E_u \times E_v) \).

We define the prefix preorder \( \preceq \) on \( \Theta_J(A) \) as the least reflexive and transitive relation satisfying for justified traces \( s \) and \( t \), \( s \preceq t \) 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 \( \sigma \) over signature \( A \), \( \sigma : A \to B \), is a nonempty and \( \preceq \)-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 \( \text{int}_J(A, B, C) \), as \( \Theta_J((A \Rightarrow B) \Rightarrow C) \). Given another signature \( D \), let \( \text{int}_J(A, B, C, D) \) 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 \( \sigma \downarrow \tau = \{ u \in \text{int}_J(A, B, C) \mid u \upharpoonright A \downarrow B \in \sigma \text{ and } u \upharpoonright B \downarrow C \in \tau \} \).

Let \( u \in \text{int}_J(A, B, C) \) 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 \( c \vartriangleleft_u b \) and \( b \vartriangleleft_u a \), then \( c \vartriangleleft_u A + C \cdot a \).

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

The result below indicates the formalism so far makes sense, allowing us to model the function space and function application.
5.1. JUSTIFIED TRACES AND CAUSAL PROCESSES

Theorem 5.10. Causal processes form a closed symmetric monoidal category, which we call 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 ⇒ A_2 satisfying, for all events e in E_u, there exists an event e' ≠ e, such that

1. e ≈_u e' and
2. [λ_u(e) = inl(a) if and only if λ_u(e') = inr(a)] and [λ_u(e) = inr(a) if and only if λ_u(e') = inl(a)]
3. (a) if λ_u(e) ∈ I_{A_1}, then e' ⊲_u e
   (b) if λ_u(e) ∈ I_{A_2}, then e ⊲_u e'
   (c) if λ_u(e) /∈ I_{A_1} and λ_u(e') /∈ I_{A_2}, then there are e_1, e_2 ∈ E_u such that e_1 ⊲_u e and e_2 ⊲_u e'
      and e_1 ≈_u e_2 and [λ_u(e_1) = inl(a') if and only if λ_u(e_2) = inr(a')] and [λ_u(e_1) = inr(a')
      if and only if λ_u(e_2) = inl(a')]

where a, a' ∈ L_A. For any two e, e' ∈ E_u, if conditions 1–3 hold, we write e ⇝_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 : Θ_I(A + B) → Δ_I(A) such that for every t = ⟨E, ≤, λ, ⊲⟩ in Θ_I(A + B)

out(t) = ⟨E', ≤', λ'⟩

where

- E' = {e ∈ E | λ(e) ∈ lin(L_A)},
CHAPTER 5. CAUSAL PROCESSES AND ASYNCHRONOUS PROCESSES

• $\preceq' = \preceq \cap (E' \times E')$,

• $\lambda' = \text{lout} \circ \lambda$ where $\text{lout}$ is the projection on labels $\text{lout} : L_A + L_B \rightarrow L_A$.

• $\trianglerighteq = \trianglerighteq \cap (E' \times E')$

We begin by showing that $\text{SynProc}_p$ 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 $\sigma : A \rightarrow B$ and $\tau : B \rightarrow C$ be causal processes and $v \in \sigma; \tau$. Take $u \in \sigma \upharpoonright \tau$ 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 $e' \trianglerighteq_v e$.

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 $e' \trianglerighteq_u e$. 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 $[s] \upharpoonright B = [s \upharpoonright B]$.

**Proof.** By inspection of the definition of trace projection and $[-]$. 

$\blacksquare$
Lemma 5.15. Composition preserves prefix-closure.

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

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

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

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

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

Let $u_2$ be defined as follows.

- $E_{u_2} = E_{v_2} + \{ e \in E_u \mid \lambda_u(e) \in L_B \text{ and } (\forall e' \in \text{last}(v_1))(e' \preceq_u e \text{ and } e \not\preceq_u e') \}$
- $\preceq_{u_2} = \preceq_u \cap (E_{u_2} \times E_{u_2})$
- $\lambda_{u_2} = \lambda_u \upharpoonright E_{u_2}$
It is clear, by construction, that \( u_1 \mid A + C = v \). Since \( \sigma_\tau \) is prefix-closed, \( u_1 \in \sigma_\tau \) 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 \( \text{SynProc}_p \), \( id_A \) 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 \( \phi : E_s \to E_{u|A+B'} \) be defined as in the first part of the proof of Lemma 3.24. We need to prove that \((\forall e, e' \in E_s)(e \triangleright_s e' \iff \phi(e) \triangleright_{u|A+B'} \phi(e'))\).

Without loss of generality, let \( E_{u|A+B} = E_s \) and \( E_{u|B'} = E_{1|B'} \). First, we show the left to right implication. Let \( e \triangleright_s e' \) which implies \( e \preceq_s e' \) and \( \lambda_s(e) \vdash_{A+B} \lambda_s(e') \). We have already shown that \( e \preceq_s e' \) iff \( \phi(e) \preceq_{u|A+B'} \phi(e') \). It is impossible for \( A \)-events to justify \( B \)-events, so we have the following three cases.

1. If \( \lambda_s(e) \in \text{inl}(L_A) \) and \( \lambda_s(e') \in \text{inl}(L_A) \), then
   
   \( \bullet \) \( \phi(e) = e \) and \( \phi(e') = e' \)
   
   \( \bullet \) \( e \triangleright_{u} e' \), \( \therefore \phi(e) \triangleright_{u|A+B'} \phi(e') \)

2. If \( \lambda_s(e) \in \text{inr}(L_B) \) and \( \lambda_s(e') \in \text{inl}(L_A) \), then
   
   \( \bullet \) \( \phi(e) \approx_e e \) and \( \phi(e') = e' \) such that \( \lambda_u(e) = \text{inl}(\text{inr}(b)) \) implies \( \lambda_u(\phi(e)) = \text{inr}(b) \), \( b \in L_B \)
   
   \( \bullet \) \( \phi(e) \triangleright_{u|B+B'} e \) because \( e \) is initial in \( B \) (using the definition of identity)
   
   \( \bullet \) \( \phi(e) \triangleright_{u|B+B'} e \) and \( e \triangleright_s e' \) implies \( \phi(e) \triangleright_{u|A+B'} \phi(e') \)

3. If \( \lambda_s(e) \in \text{inr}(L_B) \) and \( \lambda_s(e') \in \text{inr}(L_B) \), then
   
   \( \bullet \) \( \phi(e) \approx_e e \) and \( \phi(e') \approx_e e' \) such that [\( \lambda_u(e) = \text{inl}(\text{inr}(b)) \) implies \( \lambda_u(\phi(e)) = \text{inr}(b) \)]
   
   and [\( \lambda_u(e') = \text{inl}(\text{inr}(b')) \) implies \( \lambda_u(\phi(e')) = \text{inr}(b') \)], \( b, b' \in L_B \)
   
   \( \bullet \) Using the definition of identity, it follows that \( \phi(e) \triangleright_{u|B+B'} \phi(e') \).
5.1. JUSTIFIED TRACES AND CAUSAL PROCESSES

- Since $\lambda_u|_{B+B'}(\phi(e)), \lambda_u|_{B+B'}(\phi(e)) \in L_{B'}$, we have $\phi(e) \land u \phi(e')$ which implies $\phi(e) \land u \phi(e')$

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

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 $e, e' \in E_s|A$, then $e \sim s e' \iff in_1(e) \land u in_1(e')$.

**J2** If $e \in E_s|B$ and $e' \in E_s|A$, then $e \sim s e' \iff in_2(e) \land u in_1(e')$.

**J3** If $e, e' \in E_s|B$, then $e \sim s e' \iff in_2(e) \land u in_2(e') \iff in_3(e) \land u in_3(e')$.

**J4** $e \in E_s|B$ and $\lambda_s(e) \in I_B \iff in_3(e) \land u in_2(e)$.

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

- if $e \in E_s|A$ and $e' \in E_s|B$, then $[in_1(e) \land u in_2(e') \land in_1(e) \land u in_3(e')]$,
- if $e \in E_s|B$, then $in_2(e) \land u in_3(e')$,
- if $e \in E_s|B$ and $e' \in E_s|A$, then $in_3(e) \land u in_1(e')$.

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.

\[ A \rightarrow B \quad B \rightarrow B' \]
\[ \vdots \quad \vdots \]
\[ \overset{e'}{e} \quad \phi(e') \]

\[ \overset{e}{\phi(e)} \]

\[ \overset{e'}{e'} \quad \phi(e') \]

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 $int_f(A, B, C)$ and $t$ be a justified trace over $C \rightarrow D$. We have $(s \upharpoonright_{B+C})^+ t = (s^+ t) \upharpoonright_{B+C+D}$. 
Proof. First, we prove \((s \uparrow_{A+C}) \vDash t \subseteq (s \vDash t) \uparrow_{A+C+D}\). Let \(u \in (s \uparrow_{A+C}) \vDash t\); that is, \(u \in \mathit{int}_J(A, C, D)\) and \(u \upharpoonright A+C = s \upharpoonright A+C\) and \(u \upharpoonright C+D = t\). To show that \(u \in (s \vDash t) \uparrow_{A+C+D}\), we construct a trace \(v \in \mathit{int}_J(A, B, C, D)\) 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 \(\mathit{in}_J(e), j \in \{1, 2, 3, 4\}\). 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.

  \[- (\forall e, e' \in E_{u \upharpoonright A} + E_{s \upharpoonright B} + E_{u \upharpoonright C} + E_{u \upharpoonright D})(e \preceq_u e' \iff e \preceq_1 e') \]
  \[- (\forall e, e' \in E_{u \upharpoonright A} + E_{s \upharpoonright B} + E_{u \upharpoonright C})(e \preceq_s e' \iff e \preceq_1 e') \]

  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 \(e' \in E_{u \upharpoonright D}\)

  \[- (\exists e'' \in E_v)(e' \preceq_1 e'' \text{ and } e'' \preceq_1 e) \text{ if and only if } e' \preceq_2 e \]
  \[- (\exists e'' \in E_v)(e \preceq_1 e'' \text{ and } e'' \preceq_1 e') \text{ or } (\forall e'' \in E_v)((e \not\preceq_1 e'' \text{ or } e'' \not\preceq_1 e') \text{ and } (e' \not\preceq_1 e'' \text{ or } e'' \not\preceq_1 e')) \text{ if and only if } e \preceq_2 e' \]

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

- We define \(\ominus_v\) as follows. For all \(e, e' \in E_v\)

  J1 if \(e, e' \in E_{s \upharpoonright A}\), then \(e \ominus_v e' \iff e \ominus_s e' \iff e \ominus_u e'\),
  J2 if \(e, e' \in E_{s \upharpoonright B}\), then \(e \ominus_v e' \iff e \ominus_s e\),
  J3 if \(e, e' \in E_{s \upharpoonright C}\), then \(e \ominus_v e' \iff e \ominus_s e' \iff e \ominus_u e'\),
5.1. JUSTIFIED TRACES AND CAUSAL PROCESSES

\[\text{J4} \quad \text{if } e, e' \in E_u[A], \text{ then } e \sim v e' \iff e \sim u e', \]

\[\text{J5} \quad \text{if } e \in E_u[B] \text{ and } e' \in E_u[A], \text{ then } e \sim v e' \iff e \sim s e', \]

\[\text{J6} \quad \text{if } e \in E_u[C] \text{ and } e' \in E_u[B], \text{ then } e \sim v e' \iff e \sim s e', \]

\[\text{J7} \quad \text{if } e \in E_u[D] \text{ and } e' \in E_u[C], \text{ then } e \sim v e' \iff e \sim u e'.\]

\[\text{J8} \quad \text{The only pointers in } \sim v \text{ are those specified by (J1)-(J7); that is, if } e \in E_u[D] \text{ and } e' \in E_u[B] \text{ or } e \in E_u[D] \text{ and } e' \in E_u[A] \text{ or } e \in E_u[C] \text{ and } e' \in E_u[D] \text{ or } e \in E_u[C] \text{ and } e' \in E_u[B] \text{ or } e \in E_u[A] \text{ and } e' \in E_u[D] \text{ or } e \in E_u[A] \text{ and } e' \in E_u[C], \text{ then } e \not\sim v e'.\]

We have shown in the proof of Lemma 3.26 that $\preceq_\circ$ 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 $\sim v$ and the fact that $u$ and $s$ are justified.

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

We first prove that $v \mid A + B + C = s$. We have shown in the proof of Lemma 3.26 that $[v \mid A + B + C] = [s]$, i.e. $\langle E_s, \preceq_s, \lambda_s \rangle = \langle E_v[A + B + C], \preceq_v, \lambda_v \rangle$. There is a bijection $\phi : E_s \to E_v[A + B + C]$ defined as

$$\phi(e) = \begin{cases} 
\text{in}_1(e) & \text{if } \lambda_s(e) \in \text{inl}(\text{inl}(L_A)) \\
\text{in}_2(e) & \text{if } \lambda_s(e) \in \text{inl}(\text{inr}(L_B)) \\
\text{in}_3(e) & \text{if } \lambda_s(e) \in \text{inr}(L_C)
\end{cases}$$

We need to show that $(\forall e, e' \in E_s)(e \sim s e' \iff \phi(e) \sim v[A + B + C] \phi(e'))$. This follows from the definition of $\sim v$ and the fact that $s \mid A + C = u \mid A + C$.

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

The proof that $v \mid C + D = t$ follows from the following two facts: $v \mid A + C + D = u$ and $u \mid C + D = t$. Now, we prove $(s \mid_{\beta}^{A + C} t) \supseteq (s \mid_{\beta}^{A} t) \mid_{\beta}^{A + C + D}$. Let $u \in s \mid_{\beta}^{A} t$; that is, $u \in \text{int}_{\beta}(A, B, C, D)$ and $u \mid A + B + C = s$ and $u \mid C + D = t$. Applying the projection over $A + C + D$, we get $u \mid A + C + D \in \text{int}_{\beta}(A, C, D)$ and $u \mid A + C = s \mid A + C$ and $u \mid C + D = t$. So, $u \in (s \mid_{\beta}^{A + C} t)$. □
Corollary 5.18. Let \( \sigma : A \otimes B \rightarrow C \) and \( \tau : C \rightarrow D \) be causal processes. We have \( (\sigma |^A+C)_B \tau = (\sigma \tau)_B |^{A+C+D} \).

Lemma 5.19. Let \( s \) be an interaction trace over \( \text{int}_I(A, B, C, D) \). We have \( (s |^A+B+D)_C |^{A+D} = (s |^A+B)_C |^{A+D} = s |^{A+D} \).

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 \( u \in \sigma \tau \gamma \) 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 \( d \bowtie a \).

\[
A \rightarrow B \quad B \rightarrow C \quad C \rightarrow D
\]

\[
\begin{array}{cc}
    & a \\
\hline
    c & b \\
\hline
    d & c \\
\end{array}
\]

Figure 5.3: Pointer structure inheritance in Lemma 5.20

Lemma 5.20. Composition is associative.

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

By Definition 5.9, composition consists in hiding the internal channels in interaction traces and extending justification pointers. So, \( \sigma ; \tau = \sigma \tau |^A+C \).

\[
(\sigma ; \tau) ; \gamma = ((\sigma \tau)_B \gamma |^{A+C+D})_C |^{A+D} \quad \text{by Corollary 5.18}
\]

\[
= ((\sigma \tau)_B \gamma |^{A+C+D})_C |^{A+D} \quad \text{by Corollary 5.18}
\]

\[
= (\sigma \tau)_B \gamma |^{A+D} \quad \text{by Lemma 5.19}
\]

\[
= \sigma \tau \gamma |^{A+D} \quad \text{by (\star)}
\]

\[
= (\sigma \tau \gamma |^{A+B+D})_B |^{A+D} \quad \text{by Lemma 5.19}
\]

\[
= (\sigma \tau \gamma |^{B+D})_B |^{A+D} \quad \text{by Corollary 5.18}
\]

\[
= \sigma ; (\tau ; \gamma) \quad \text{by Corollary 5.18}
\]
The following Proposition sums up our results so far.

**Proposition 5.21.** SynProc is a category.

Next, we demonstrate that SynProc 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 $\text{id}_A \otimes \text{id}_B = \text{id}_{A \otimes B}$.

First, we prove $\text{id}_A \otimes \text{id}_B \subseteq \text{id}_{A \otimes B}$. Let $u \in \text{id}_A \otimes \text{id}_B$; that is, $u \in \Theta_J(A \otimes B \Rightarrow A' \otimes B')$ and $u \upharpoonright A + A' \in \text{id}_A$ and $u \upharpoonright B + B' \in \text{id}_B$. Using Definition 5.11, we get $u \in \Theta_J(A \otimes B \Rightarrow A' \otimes B')$ and $(\forall e \in E_{u|A+A'}, \exists e' \in E_{u|A+A'})(e \approx_{u|A+A'} e')$ and $(\forall e \in E_{u|B+B'}, \exists e' \in E_{u|B+B'})(e \approx_{u|B+B'} e')$. Since $E_u = E_{u|A+A'} \cup E_{u|B+B'}$ and $(\forall e, e' \in E_u)(e \approx_{u|A+A'} e' \lor e \approx_{u|B+B'} e')$, then $e \approx_{u'} e'$ and for all $a \in E_{u|A+A'}, b \in E_{u|B+B'}$, we have $\lambda_{u|A+A'}(a) = \lambda_u(a)$ and $\lambda_{u|B+B'}(b) = \lambda_u(b)$, we conclude that for all $e \in E_u$, there is $e' \in E_u$ such that $e \approx_{u'} e'$. Therefore, $u \in \text{id}_{A \otimes B}$.

Now, we prove $\text{id}_A \otimes \text{id}_B \supseteq \text{id}_{A \otimes B}$. Let $u \in \text{id}_{A \otimes B}$; that is, $u \in \Theta_J(A \otimes B \Rightarrow A' \otimes B')$ and $(\forall e \in E_u, \exists e' \in E_u)(e \approx e')$. After projecting $u$ over $A + A'$, we get $u \upharpoonright A + A' \in \Theta_J(A \Rightarrow A')$ and $(\forall e \in E_{u|A+A'}, \exists e' \in E_{u|A+A'})(e \approx_{u|A+A'} e')$; that is, $u \upharpoonright A + A' \in \text{id}_A$. Projecting $u$ over $B + B'$ yields $u \upharpoonright B + B' \in \Theta_J(B \Rightarrow B')$ and $(\forall e \in E_{u|B+B'}, \exists e' \in E_{u|B+B'})(e \approx_{u|B+B'} e')$; that is, $u \upharpoonright B + B' \in \text{id}_B$. So, $u \in \text{id}_A \otimes \text{id}_B$.

Note that, in signature $A \otimes B \Rightarrow C \otimes D$, initial $D$-events may cause initial $A$-events. However, in $\text{id}_{A \otimes B}$, 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 $(\sigma^A \gamma^B) \otimes \tau = (\sigma^A \gamma) \otimes \tau^B_{A+C+D+E}$. 

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,

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

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 $(A \otimes B) \otimes C \rightarrow A \otimes (B \otimes C)$ 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 $\alpha$. 

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 \( A \in \text{SynProc}_P \), \( I \otimes A \) and \( A \otimes I \) are clearly isomorphic to \( A \) as tensoring with \( I \) only retags its elements. Therefore, we define the left identity isomorphism \( \lambda_A \) and the right identity isomorphism \( \rho_A \) as those with the signatures above, and which consist of exactly the same traces, up to retagging of labels by \( \text{inl} \) and \( \text{inr} \), as \( \text{id}_A \). The coherence condition in Figure 3.2 directly follows from the definitions of the tensor product, \( \lambda \), and \( \rho \). ■

Next, we show that \( \text{SynProc}_P \) is symmetric.

Proposition 5.28. \( \text{SynProc}_P \) 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 \( \gamma_{A,B}; \gamma_{B,A} = \text{id}_{A \otimes B} \).

Proof. For any \( A, B \) in \( \text{SynProc}_P \), 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 \( \text{inl} \) and \( \text{inr} \), as \( \text{id}_{A \otimes B} \). 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 \( \text{inl} \) and \( \text{inr} \), as \( \text{id}_{A \otimes B}; \text{id}_{A \otimes B} \). However, as \( \gamma_{A,B} \) changes the tags of labels and \( \gamma_{B,A} \) changes them back according to \((\ast)\), \( \gamma_{A,B}; \gamma_{B,A} = \text{id}_{A \otimes B}; \text{id}_{A \otimes B} = \text{id}_{A \otimes B} \).

Finally, we show that \( \text{SynProc}_P \) is closed.
CHAPTER 5. CAUSAL PROCESSES AND ASYNCHRONOUS PROCESSES

Proposition 5.29. SynProc\textsubscript{p} with $- \otimes -$, \textit{I}, \text{eval}_{A,B}$ is a closed symmetric monoidal category.

\textbf{Proof.} We need to show that \textit{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 $\sigma = (\tau \otimes \text{id}_B) ; \text{eval}_{B,C}$. It can be easily verified that $\tau$ with the same set of traces as $\sigma$ satisfies the property. It is also clear that $\tau$ is unique: its traces have to be the same as those of $\sigma$ since $\text{eval}_{B,C}$ is isomorphic to $\text{id}_B \otimes \text{id}_C$ and therefore, $(- \otimes \text{id}_B) ; \text{eval}_{B,C}$ only retags the events in the traces of any morphism placed in the hole $(-)$ according to the following.

\begin{align*}
L_{A \otimes B = C} &= \text{inl}(\text{inl}(A)) \cup \text{inl}(\text{inr}(B)) \cup \text{inr}(C) \\
L_{A \Rightarrow B = C} &= \text{inl}(A) \cup \text{inr}(\text{inl}(B)) \cup \text{inr}(\text{inr}(C))
\end{align*}

\[\square\]

5.2 Asynchronous Processes

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

\textbf{Definition 5.30 (Asynchronous trace).} A justified trace $\langle E, \preceq, \lambda, \cdot \rangle$ 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 $\text{pos}_t : \{1, \ldots, n\} \to E_t$ satisfying $(\forall e, e' \in E_t)(e \preceq_t e' \Leftrightarrow \text{pos}_t^{-1}(e) < \text{pos}_t^{-1}(e'))$.

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.

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

1. For all \( e_1, e_2 \in E_s \setminus \{e\} \), we have \( e_1 \leq_s e_2 \) if and only if \( \phi(e_1) \leq_s \phi(e_2) \).

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

3. For all \( e_1, e_2 \in E_s \), we have \( e_1 \prec_s e_2 \) if and only if \( \phi(e_1) \prec_s \phi(e_2) \).

4. (a) \( e \) is an input and \( \text{pos}_s(e) \leq \text{pos}_{s'}(\phi(e)) \), or

(b) \( e \) is an output and \( \text{pos}_s(e) \leq \text{pos}_{s'}(\phi(e)) \).

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

This preorder is a reformulation of a saturation principle which has long been used to model asynchronous systems [Udd86, JIH90]. 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 \( \preceq^{sm} \) 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 \( \preceq^0 \) be the relation on \( \Gamma(A) \) satisfying \( s' \preceq^0 s \) 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. \( (\forall e, e' \in E_s)(e \prec_s e' \text{ if and only if } \phi(e) \prec_s \phi(e')) \)

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 \( s' \preceq^0 s \) if and only if \( s' \preceq s \).

**Proof.** First, we show the left to right implication. Let \( s' \preceq^0 s \). 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 \( \prec_s = \prec_{s'} \). We show that if \( |E_s| = n \), there exists a series of \( n - 1 \) permutations of \( s \) such that \( s' = s_n \preceq s_{n-1} \preceq \ldots \preceq s_1 \preceq s_0 = s \). 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 $A_{k+1} = \{ e \in E_{s_k} \mid pos_{s_k}(k + 1) \preceq_{s_k} e \text{ and } e \neq pos_{s'}(k + 1) \}$. } 

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.
  - $E_{s_{k+1}} = E_{s_k}$
  - $\lambda_{s_{k+1}} = \lambda_{s_k}$
  - $\mathcal{S}_{s_{k+1}} = \mathcal{S}_{s_k}$
  - $\preceq_{s_{k+1}}$ is defined as follows, for all $e, e' \in E_{s_k}$.
    - $\text{P1}$ If $e, e' \in B_{k+1}$, then $e \preceq_{s_k} e' \iff e \preceq_{s_{k+1}} e'$.
    - $\text{P2}$ If $e, e' \in A_{k+1}$, then $e \preceq_{s_k} e' \iff e \preceq_{s_{k+1}} e'$.
    - $\text{P3}$ If $e \in B_{k+1}$ and $e' \in A_{k+1}$, then $e \preceq_{s_{k+1}} e'$.
    - $\text{P4}$ If $e \in B_{k+1}$, then $e \preceq_{s_{k+1}} pos_{s'}(k + 1)$.
    - $\text{P5}$ If $e \in A_{k+1}$, then $pos_{s'}(k + 1) \preceq_{s_{k+1}} e$.
    - $\text{P6}$ $pos_{s'}(k + 1) \preceq_{s_{k+1}} pos_{s'}(k + 1)$

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 $(\forall e, e' \in E_{s_{k+1}})(e \mathcal{S}_{s_{k+1}} e' \Rightarrow e \preceq_{s_{k+1}} e')$. We have $e \mathcal{S}_{s_{k+1}} e' \iff e \mathcal{S}_{s_k} e'$. 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).
5.2. ASYNCHRONOUS PROCESSES

If \( e = \text{pos}_{s}(k+1) \) 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 \( \text{pos}_{s_{k}}(k+1) \preceq_{s_{k}} e \) and \( e' \preceq_{s_{k}} \text{pos}_{s_{k}}(k) \) 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 \( e \prec_{s_{k}} e' \).

If \( e = \text{pos}_{s}(k+1) \) and \( e' \in B_{k+1} \), then \( e' \preceq_{s_{k}} \text{pos}_{s_{k}}(k) \). Since \( \text{pos}_{s_{k}}(k) \preceq_{s_{k}} \text{pos}_{s_{k}}(k+1) \), we get \( e' \preceq_{s_{k}} e \). We then follow the same reasoning as the previous case.

If \( e \in A_{k+1} \) and \( e' = \text{pos}_{s}(k+1) \), then \( \text{pos}_{s_{k}}(k+1) \preceq_{s_{k}} e \) and \( e \neq \text{pos}_{s}(k+1) \). We have \( e \prec_{s_{k}} e' \). 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 \( b \not\preceq a \).

**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 \( pos_{s_{k+1}}(k+1) < pos_{s_{k+1}}^{-1}(e) \) and \( e \prec_{s_{k+1}} e' \). Since \( pos_{s_{k}}(k+1) \prec_{s_{k+1}} e' \). 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 \( e_{0} \not\preceq_{s'} pos_{s_{k+1}}(k+1) \). This contradicts \( s' \preceq_{s} s \).

We can now verify that \( s_{k+1} \preceq_{s} s_{k} \). 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 $\precsim s_m$. If it is an output, then all events in $E_0 = \{ \text{pos}_{s_{k+1}}(i) \mid k + 1 \leq i < x \}$ are outputs by Fact 2 and therefore making all events in $E_0$ later than $\text{pos}_{s_{k+1}}(k + 1)$ is consistent with $\precsim$; that is, if $|E_0| = m$, there are $m$ traces $s_j$, $j \in \{1, \ldots, m\}$ such that $s_{k+1} = s_m \precsim s_{m-1} \precsim \ldots \precsim s_1 \precsim s_k$.

Now, we prove the right to left implication. Let $s' \precsim s$. 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 $\curvearrowright_s = \curvearrowright_{s'}$. From $s' \precsim s$, we know that there are $n$ traces $s_i$, $i \in \{1, \ldots, n\}$ such that $s' = s_n \precsim s_{n-1} \precsim \ldots \precsim s_1 \precsim s_0 = s$. 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 $e \precsim s_i$ and $e$ is an input and $e'$ is an output, then $e \precsim s'_{i+1} e'$. This follows from Definition 5.31.

In the sequel, we use the preorders $\precsim$ and $\precsim^0$ 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 $[t] = [s] \cdot i$, where $i$ is an input, then $t \in \sigma$.

It is convenient to define an $O$-extension preorder $\precsim$ on $\Gamma(A)$ as the least reflexive and transitive relation satisfying for asynchronous traces $s$ and $t$, $t \precsim s$ if $s$ is a prefix of $t$ and $[t] = [s] \cdot i$, 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 $\precsim$-downward closed set of asynchronous traces.

Processes that are $\precsim$-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 $\text{int}_A(A, B, C)$ 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 $\text{SynProc}_p$, 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 $\text{cc}_A$, is the set of traces $u \in \Gamma^{\text{alt}}(A \Rightarrow A')$ 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 $e \in E_u$,

- $\text{iccopy}_u(e) \preceq_u e$ and there is no $e' \in E_u$ satisfying $e' \neq e$ and $e' \neq \text{iccopy}_u(e)$ and $\text{iccopy}_u(e) \preceq_u e'$ and
- $\lambda_u(\text{iccopy}_u(e)) = \text{inl}(a)$ iff $\lambda_u(e) = \text{inr}(a)$ and $\lambda_u(\text{iccopy}_u(e)) = \text{inr}(a)$ iff $\lambda_u(e) = \text{inl}(a)$

- if $\lambda_u(e) \in I_A$ or $\lambda_u(\text{iccopy}_u(e)) \in I_{A'}$, then $\text{iccopy}_u(e) \preceq_u e$
- 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 $e_1 \preceq_u e$ and $e_2 \preceq_u \text{iccopy}_u(e)$ and $e_1 = \text{iccopy}_u(e_2)$.

Note that for each $u \in \Gamma^{\text{alt}}(A \Rightarrow A')$, if $\text{iccopy}_u$ 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

\[
icopy_u : \{ e \in E_u \mid \pi \circ \lambda_u(e) = o \} \to \{ e \in E_u \mid \pi \circ \lambda_u(e) = i \}
\]

assigning to each output an input copy such that for each output \( e \in E_u \),

\[
\begin{align*}
I1 & \quad \nicopy_u(e) \preceq_u e \text{ and} \\
I2 & \quad \lambda_u(\nicopy_u(e)) = \text{inl}(a) \iff \lambda_u(e) = \text{inr}(a) \text{ and } \lambda_u(\nicopy_u(e)) = \text{inl}(a) \\
& \text{ and} \\
I3 & \quad \bullet \text{ if } \lambda_u(e) \in I_A \text{ or } \lambda_u(\nicopy_u(e)) \in I_{A'}, \text{ then } \nicopy_u(e) \lhd_u e \\
& \quad \bullet \text{ if } \lambda_u(e) \notin I_A \text{ and } \lambda(\nicopy_u(e)) \notin I_{A'}, \text{ then there are } e_1, e_2 \in E_u \text{ such that } e_1 \lhd_u e \text{ and } e_2 \lhd_u \nicopy_u(e) \text{ and } e_1 = \nicopy_u(e_2).
\end{align*}
\]

Note that the inverse of \( \nicopy_u \) is a partial function, denoted by \( \text{ocopy}_u \), that maps inputs to their output copies.

**Theorem 5.38.** Asynchronous processes form a closed symmetric monoidal category that we call \( \text{AsyProc} \).

The proof is detailed in the next section.

### 5.2.1 A Category of Asynchronous Processes

Intuitively, the category \( \text{AsyProc} \) is equivalent to the full subcategory of \( G \)—the category of multi-threaded saturated strategies \([\text{GM08, pp. 14–18}]\)—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 \([\text{GM08}]\). We adopt the same strategy here. We need to show that if \( v' \preceq v \) and \( v \in \sigma; \tau \), then \( v' \in \sigma; \tau \). By Definition \[5.31\] we know that \( v' \preceq v \) implies...
there are traces \( v_i, i \in \{1, \ldots, n-1\} \) such that \( v' = v_n \preceq_{\text{sm}} v_{n-1} \preceq_{\text{sm}} \ldots \preceq_{\text{sm}} v_1 \preceq_{\text{sm}} v \). So, it is sufficient to show: if \( v' \preceq_{\text{sm}} v \) 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 \( u = u_1 \cdot e \cdot u_B \cdot i \cdot u_2 \in \sigma; \tau \) satisfying \( u \upharpoonright A + C = v \) and \( u_1 \upharpoonright A + C = v_1 \) and \( u_2 \upharpoonright A + C = v_2 \). Making \( i \) earlier in \( u_B \) still results in interaction traces. This is possible because \( \sigma \) and \( \tau \) are saturated. So, if \( i \) is a C-event we call upon \( \tau \) and if it is an A-event we use \( \sigma \). 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, \( u = u_1 \cdot e \cdot i \cdot u_B \cdot u_2 \in \sigma; \tau \). At this point, if both \( e \) and \( i \) are A-events (respectively C-events) and \( e \not\wedge i \), then by the saturation of \( \sigma \) (respectively \( \tau \)), we get \( u_1 \cdot i \cdot e \cdot u_B \cdot u_2 \in \sigma; \tau \). Otherwise, \( u_1 \cdot i \cdot e \cdot u_B \cdot u_2 \in \sigma; \tau \), 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 \( u = u_1 \cdot o \cdot u_B \cdot e \cdot u_2 \in \sigma; \tau \) satisfying \( u \upharpoonright A + C = v \) and \( u_1 \upharpoonright A + C = v_1 \) and \( u_2 \upharpoonright A + C = v_2 \). Making \( o \) later in \( u_B \) still results in interaction traces. This is possible because \( \sigma \) and \( \tau \) are saturated. So, if \( o \) is a C-event we call upon \( \tau \) and if it is an A-event we use \( \sigma \). 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, \( u = u_1 \cdot u_B \cdot o \cdot e \cdot u_2 \in \sigma; \tau \). At this point, if both \( o \) and \( e \) are A-events (respectively C-events) and \( o \not\wedge e \), then by the saturation of \( \sigma \) (respectively \( \tau \)), we get \( u_1 \cdot u_B \cdot e \cdot o \cdot u_2 \in \sigma; \tau \). Otherwise, \( u_1 \cdot u_B \cdot e \cdot o \cdot u_2 \in \sigma; \tau \), 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 \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \) 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 \( u \in \sigma; \tau \) such that \( u \upharpoonright A + C = v \). So, \( u \upharpoonright A + B \in \sigma \) and \( u \upharpoonright B + C \in \tau \).
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 \( e' \cap_w i \) and \( \lambda_w(e') \vdash_{A \Rightarrow C} \lambda_w(i) \). If \( i \) is a \( C \)-event then
   \[
   (u \mid B + C) \cdot i \in \Gamma(B \Rightarrow C)
   \]
   because \( e' \in E_v \iff e' \in E_{u|B+C} \) and \( e' \cap_w i \iff e' \cap_{u|B+C} i \)
   for \( \lambda_w(e') \vdash_{A \Rightarrow C} \lambda_w(i) \). If \( i \) is a \( B \)-event, then
   \[
   \lambda_w(e') \vdash_{A \Rightarrow C} \lambda_w(i) \iff \lambda_{u|B+C}(i') \vdash_{B \Rightarrow C} \lambda_{u|B+C}(i)
   \]
   because \( \tau \) is \( O \)-complete, we have \( \lambda_w(e') \vdash_{A \Rightarrow C} \lambda_w(i) \). So, \( u \cdot i \in \sigma \frac{1}{2} \tau \). Finally, we get \( \nu \cdot i \in \sigma; \tau \). 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 \( \sigma \) is \( O \)-complete.
   - If \( e' \) is a \( C \)-event, then, by definition of interaction, there is a \( B \)-event \( b \) such that
     \[
     e' \cap_{u|B+C} b \quad \text{and} \quad b \cap_{u|A+B} i
     \]
     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 \( \sigma \) is \( O \)-complete. If \( i \) is an \( C \)-event, we use the fact that \( \tau \) is \( O \)-complete.

Most required proofs are exactly the same as those we presented for SynProc. 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 \( C \rightarrow D \). We have \( (s \upharpoonright_{B+C}) \frac{A+C}{B} t = (s \frac{A+C}{C}) t \upharpoonright_{A+C+D} \).

**Proof.** First, we prove \( (s \upharpoonright_{B+C}) \frac{A+C}{B} t \subseteq (s \frac{A+C}{C}) t \upharpoonright_{A+C+D} \). Let \( u \in (s \upharpoonright_{B+C}) \frac{A+C}{B} t \); that is, \( u \in int_A(A, C, D) \) and \( u \mid A + C = s \mid A + C \) and \( u \mid C + D = t \). To show that \( u \in (s \frac{A+C}{C}) t \upharpoonright_{A+C+D} \), we construct a trace \( v \in int_A(A, B, C, D) \) such that \( v \mid A + C + D = u \) and \( v \mid A + B + C = s \) and \( v \mid C + D = t \) as follows.

Without loss of generality, we assume that \( E_{u|A+C} = E_{u|A+C} \). 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:

- \( E_v = E_{u|A} + E_{u|B} + E_{u|C} + E_{u|D} \). 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 \(E_{u|A}, E_{s|B}, E_{u|C}\) and \(E_{u|D}\) are disjoint. It should improve readability at the cost of abusing notation.

- \(\lambda_v = \lambda_{u|A} + \lambda_{s|B} + \lambda_{u|C} + \lambda_{u|D}\)

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

  \[\begin{align*}
  &P1 \quad (\forall e, e' \in E_{u|A} + E_{u|C} + E_{u|D})(e \preceq_u e' \Leftrightarrow e \preceq_1 e') \\
  &P2 \quad (\forall e, e' \in E_{u|A} + E_{s|B} + E_{u|C})(e \preceq_s e' \Leftrightarrow e \preceq_1 e')
  \end{align*}\]

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 \(e \in E_{s|B}\) and \(e' \in E_{u|D}\)

- \(P3 \quad (\exists e'' \in E_v)(e' \preceq_1 e'' \text{ and } e'' \preceq_1 e)\) if and only if \(e' \preceq_2 e\)
- \(P4 \quad (\exists e'' \in E_v)(e \preceq_1 e'' \text{ and } e'' \preceq_1 e')\) or \((\forall e'' \in E_v)((e \preceq_1 e'' \text{ or } e'' \preceq_1 e') \text{ and } (e' \preceq_1 e'' \text{ or } e'' \preceq_1 e))\) 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 \(\blacktriangleleft_v\) as follows. For all \(e, e' \in E_v\),

  \[\begin{align*}
  &J1 \quad \text{if } e, e' \in E_{s|A}, \text{ then } e \blacktriangleleft_v e' \Leftrightarrow e \blacktriangleleft_s e' \Leftrightarrow e \blacktriangleleft u e'. \\
  &J2 \quad \text{if } e, e' \in E_{s|B}, \text{ then } e \blacktriangleleft_v e' \Leftrightarrow e \blacktriangleleft_s e. \\
  &J3 \quad \text{if } e, e' \in E_{s|C}, \text{ then } e \blacktriangleleft_v e' \Leftrightarrow e \blacktriangleleft_s e' \Leftrightarrow e \blacktriangleleft u e'. \\
  &J4 \quad \text{if } e, e' \in E_{s|D}, \text{ then } e \blacktriangleleft_v e' \Leftrightarrow e \blacktriangleleft_u e'. \\
  &J5 \quad \text{if } e \in E_{s|B} \text{ and } e' \in E_{s|A}, \text{ then } e \blacktriangleleft_v e' \Leftrightarrow e \blacktriangleleft_s e'. \\
  &J6 \quad \text{if } e \in E_{s|C} \text{ and } e' \in E_{s|B}, \text{ then } e \blacktriangleleft_v e' \Leftrightarrow e \blacktriangleleft_s e'. \\
  &J7 \quad \text{if } e \in E_{u|D} \text{ and } e' \in E_{u|C}, \text{ then } e \blacktriangleleft_v e' \Leftrightarrow e \blacktriangleleft_u e'. \\
  &J8 \quad \text{The only pointers in } \blacktriangleleft_v \text{ are those specified by (J1)–(J7); that is, if } [e \in E_{u|D} \text{ and } e' \in E_{s|B}] \text{ or } [e \in E_{u|D} \text{ and } e' \in E_{s|A}] \text{ or } [e \in E_{s|C} \text{ and } e' \in E_{u|D}] \text{ or } [e \in E_{s|C} \text{ and } e' \in E_{s|A}] \text{ or } [e \in E_{s|B} \text{ and } e' \in E_{u|D}] \text{ or } [e \in E_{s|B} \text{ and } e' \in E_{s|C}] \text{ or } [e \in E_{s|A} \text{ and } e' \in E_{u|D}] \text{ or } [e \in E_{s|A} \text{ and } e' \in E_{s|B}], \text{ then } e \blacktriangleleft_v e'.
  \end{align*}\]

We have shown in the proof of Lemma 3.26\footnote{Page 109} 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\footnote{Page 109}.
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 \( (A, A) \), \( (A, B) \), \( (A, C) \), \( (A, D) \), \( (B, A) \), \( (B, B) \), \( (B, C) \), \( (C, A) \), \( (C, B) \), \( (C, C) \), \( (C, D) \), \( (D, A) \), \( (D, D) \) are similar, so we only prove it for the case where \( \lambda_v(e), \lambda_v(e') \in L_A \). We have \( e \preceq_v e' \iff e \preceq_1 e' \) by \((P1)\). Similarly, \( e' \preceq_v e \iff e' \preceq_1 e \iff e' \preceq_s e \) 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

- \( e \preceq_v e' \iff e \preceq_2 e' \) iff
  - \( (\exists e_0 \in E_v)(e \preceq_1 e_0 \text{ and } e_0 \preceq_1 e') \) (let us refer to this by \((\dagger)\)) or
  - \( (\forall e'' \in E_v)((e \npreceq_1 e'' \text{ or } e'' \npreceq_1 e') \text{ and } (e' \npreceq_1 e'' \text{ or } e'' \npreceq_1 e')) \) (let us refer to this by \((\ddagger)\))

- \( e' \preceq_v e \iff e' \preceq_2 e \iff (\exists e_1 \in E_v)(e' \preceq_1 e_1 \text{ and } e_1 \preceq_1 e) \) (let us refer to this by \((\circ)\))

We have \((\dagger)\) and \((\circ)\) is a contradiction. Suppose \((\ast)\) and \((\ddagger)\). We have \((\ast) \iff (\exists e_0 \in E_v)(e \preceq_s e_0 \text{ and } e_0 \preceq u e') \) and \((\ddagger) \iff (\exists e_1 \in E_v)(e' \preceq u e_1 \text{ and } e_1 \preceq s e) \). We also have \( e_0, e_1 \in E_{u \mid A + C} = E_{s \mid A + C} \). 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 \( s' \preceq s \); 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** \( (\forall e, e' \in E_s)(e \lhd_{s'} e' \iff \text{if and only if } \phi(e) \lhd_{s'} \phi(e')) \)

**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 \( \icopy_{s'}(e) = \phi(\icopy_s(\phi^{-1}(e))) \). We need to prove that \( s' \) with \( \icopy_{s'} \) satisfies conditions (I1)–(I3) of Definition 5.37. Take an output \( e \) in \( s' \) and let \( e' = \icopy_s(\phi^{-1}(e)) \). So, \( \icopy_{s'}(e) = \phi(e') \).

- We first show that \( \icopy_{s'}(e) \preceq_{s'} e \). We have by Definition 5.37 \( \icopy_s(\phi^{-1}(e)) \preceq_s \phi^{-1}(e) \).

So, \( e' \preceq \phi^{-1}(e) \). 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, \( \icopy_{s'}(e) \preceq_{s'} e \).

- We have \( \lambda_{s'}(e') = \text{inl}(a) \iff \lambda_s(\phi^{-1}(e)) = \text{inr}(a), a \in L_A \). By (S1), \( \lambda_s(e') = \text{inl}(a) \iff \lambda_{s'}(\phi(e')) = \text{inl}(a) \) and \( \lambda_s(\phi^{-1}(e)) = \text{inr}(a) \iff \lambda_{s'}(e) = \text{inr}(a) \). So, \( \lambda_{s'}(\phi(e')) = \text{inl}(a) \iff \lambda_{s'}(e) = \text{inr}(a) \).

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 \( e' \lhd_{s'} \phi^{-1}(e) \). We then use (S2) to find \( \phi(e') \lhd_{s'} e \).

- Let \( \lambda_{s'}(\phi(e')) \not\in I_{A'} \) and \( \lambda_{s'}(e) \not\in I_A \). By (S1), we get \( \lambda_s(e') \not\in I_{A'} \) and \( \lambda_s(\phi^{-1}(e)) \not\in I_A \). 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 \( e_1 \lhd_{s} e' \) and \( e_2 \lhd_{s} \phi^{-1}(e) \) and \( e_2 = \icopy_s(e_1) \). We then use (S2) to find \( \phi(e_1) \lhd_{s'} \phi(e') \) and \( \phi(e_2) \lhd_{s'} e \) and \( \icopy_{s'}(\phi(e_1)) = \phi(\icopy_s(\phi^{-1}(\phi(e_1)))) = \phi(\icopy_s(e_1)) = \phi(e_2) \).

**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 \( u \in \sigma A \) and \( u \mid B + B' \) has even length, then \( u \mid A + B' \preceq^0 u \mid A + B \).

**Proof.** Let \( u \in \sigma A \) and \( u \mid B + B' \) has even length. Following Definition 5.32, we first need to provide a bijection \( \phi : E_{u|A+B} \to E_{u|A+B'} \) such that \( \lambda_{u|A+B} = \lambda_{u|A+B'} \circ \phi \) and \( (\forall e, e' \in E_{u|A+B})(e \lhd_{u|A+B} e' \iff \text{if and only if } \phi(e) \lhd_{u|A+B'} \phi(e')) \).
Define $ccopy = iccopy_{u|B+B'} \cup iccopy_{u|B+B'}^{-1}$. Note that the domains of $iccopy_{u|B+B'}$ and $iccopy_{u|B+B'}^{-1}$ are disjoint and so are their codomains. Moreover, the domain and codomain of $ccopy$ is equal to $E_{u|B+B'}$. By Definition 5.36 (copycat), we know that the input copy of each output in $u \upharpoonright B + B'$ is the event that directly precedes it. Since $u \upharpoonright B + B'$ has even length, its last event is an output, and so each input has an output copy. Therefore, $iccopy_{u|B+B'}^{-1}$ is an injection. So, copycat is a bijection. Additionally, note that copycat is an involution.

Define $\phi : E_{u|A+B} \to E_{u|A+B'}$ as follows, for all $e \in E_{u|A+B}$.

$$
\phi(e) = \begin{cases} 
e e & \text{if } \lambda_u(e) \in L_A \\
ccopy(e) & \text{if } \lambda_u(e) \in L_B.
\end{cases}
$$

Its inverse $\phi^{-1}$ is defined as follows, for all $e \in E_{u|A+B'}$.

$$
\phi^{-1}(e) = \begin{cases} 
e e & \text{if } \lambda_u(e) \in L_A \\
ccopy(e) & \text{if } \lambda_u(e) \in L_B.
\end{cases}
$$

It is clear that $\phi$ is a bijection.

We first prove that $\lambda_{u|A+B} = \lambda_{u|A+B'} \circ \phi$. Let $e \in E_{u|A+B}$. If $\lambda_{u|A+B}(e) \in L_A$, then $\phi(e) = e$. We have $\lambda_{u|A+B}(e) = a \in L_A \implies \lambda_u(e) = a \implies \lambda_{u|A+B'}(\phi(e)) = a$. If $\lambda_{u|A+B}(e) \in L_B$, then $\phi(e) = ccopy(e)$. We have $\lambda_{u|A+B}(e) = b \in L_B \implies \lambda_u(e) = inl(inr((b))) \implies \lambda_{u|B+B'}(e) = inl(b)$. Using Definition 5.36 (copycat), we get $\lambda_{u|B+B'}(ccopy(e)) = inr(b) \implies \lambda_u(\phi(e)) = inr(b) \implies \lambda_{u|A+B'}(\phi(e)) = b$.

Next we show that $(\forall e, e' \in E_{u|A+B})(e \cap_{u|A+B} e' \iff \phi(e) \cap_{u|A+B'} \phi(e'))$. For the left to right implication, suppose $e, e' \in E_{u|A+B}$ and $e \cap_{u|A+B} e'$. 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 $e \cap_{u|A+B} e' : e \cap_{u} e' : e \cap_{u|A+B'} \phi(e')$.

- If $\lambda_u(e), \lambda_u(e') \in L_B$, then $\phi(e) = ccopy(e)$ and $\phi(e') = ccopy(e')$. We have $e \cap_{u|A+B} e' : e \cap_{u} e' : e \cap_{u|B+B'} e'$. We then have the following two cases.

  - Let $e$ be an input and $e'$ be an output in $u \upharpoonright B + B'$. Using Definition 5.36, $e \cap_{u|B+B'} e'$. For the right to left implication, suppose $e, e' \in E_{u|A+B}$ and $e \cap_{u|A+B} e'$. We have the following cases.
5.2. ASYNCHRONOUS PROCESSES

For the right to left implication, suppose \( e \in E_{u | A + B} \) and \( \phi(e) \setminus u | A + B \) \( \phi\). So, \( \text{ccopy}(e) \setminus u | B + B' \) \( \text{ccopy}(e) \setminus u | A + B' \) \( \phi(e) \setminus u | A + B' \) \( \phi\).

- Let \( e \in E_{u | A + B} \) and \( \phi(e) \setminus u | B + B' \) \( e' \) be an input in \( u \cup B + B' \). Using Definition 5.36, \( e \setminus u | B + B' \) \( e' \) : \( \text{iccopy}(e) \setminus u | B + B' \) \( \text{iccopy}(e) \setminus u | A + B' \) \( \phi(e) \setminus u | A + B' \) \( \phi(e) \setminus u | A + B' \) \( \phi\).

- If \( \lambda_u(e) \in L_B \) and \( \lambda_u(e') \in L_A \), then \( \phi(e) = \text{ccopy}(e) \) and \( \phi(e') = e' \). From \( e \setminus u | A + B \) \( e' \) : \( \text{iccopy}(e) \setminus u | B + B' \) \( \text{iccopy}(e) \setminus u | A + B' \) \( \phi(e) \setminus u | A + B' \) \( \phi(e) \setminus u | A + B' \) \( \phi\).

  - If \( \lambda_u(e), \lambda_u(e') \in L_A \), then \( \phi(e) = \text{ccopy}(e) \) and \( \phi(e') = e' \). We have \( \phi(e) \setminus u | A + B' \) \( \phi(e') \) : \( \phi(e) \setminus u | A + B' \) \( \phi(e') \) : \( e \setminus u | A + B' \) \( e' \).

  - If \( \lambda_u(e), \lambda_u(e') \in L_B \), then \( \phi(e) = \text{ccopy}(e) \) and \( \phi(e') = \text{ccopy}(e') \). We have \( \phi(e) \setminus u | A + B' \) \( \phi(e') \) : \( \phi(e) \setminus u | A + B' \) \( \phi(e') \) : \( \phi(e) \setminus u | A + B' \) \( \phi(e') \). We then have the following two cases.

    - Let \( \phi(e) \) be an input and \( \phi(e') \) be an output in \( u \cup B + B' \). Using Definition 5.36, \( \phi(e) \setminus u | B + B' \) \( \phi(e') \) : \( \text{iccopy}(\phi(e)) \setminus u | B + B' \) \( \text{iccopy}(\phi(e')) \) : \( \phi(e) \setminus u | B + B' \) \( \phi(e') \) : \( \phi(e) \setminus u | B + B' \) \( \phi(e') \).

    - Let \( \phi(e) \) be an output and \( \phi(e') \) be an input in \( u \cup B + B' \). Using Definition 5.36, \( \phi(e) \setminus u | B + B' \) \( \phi(e') \) : \( \text{iccopy}(\phi(e)) \setminus u | B + B' \) \( \text{iccopy}(\phi(e')) \) : \( \phi(e) \setminus u | B + B' \) \( \phi(e') \) : \( \phi(e) \setminus u | B + B' \) \( \phi(e') \).

  - If \( \lambda_u(e) \in L_B \) and \( \lambda_u(e') \in L_A \), then \( \phi(e) = \text{ccopy}(e) \) and \( \phi(e') = e' \). From \( \phi(e) \setminus u | A + B' \) \( \phi(e') \), 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 \( \phi(e) \setminus u | B + B' \) \( e_0 \) and \( e_0 \setminus u | A + B' \) \( e' \). By Definition 5.36, \( \phi(e) \setminus u | B + B' \) \( \text{iccopy}(\phi(e)) \setminus u | B + B' \) \( \text{iccopy}(\phi(e')) \) : \( \phi(e) \setminus u | B + B' \) \( \phi(e') \) : \( \phi(e) \setminus u | B + B' \) \( \phi(e') \).

Now, we show that the third condition in Definition 5.32 holds, i.e. for all \( e, e' \in E_{u | A + B} \) if \( e \preceq_{u | A + B} e' \) and \( e \) is an input and \( e' \) is an output, then \( \phi(e) \preceq_{u | A + B} \phi(e') \). Let \( e \preceq_{u | A + B} e' \) and \( e \) be an input and \( e' \) be an output. We have the following cases.
If \( e, e' \in E_u|A \), then \( \phi(e) = e \) and \( \phi(e') = e' \). We have \( e \preceq_{u|A+B} e' : e \preceq_{u} e' : e \preceq_{u|A+B'} e' : \phi(e) \preceq_{u|A+B'} \phi(e') \).

- If \( e, e' \in E_u|B \), then \( \phi(e) = \text{ccopy}(e) \) and \( \phi(e') = \text{ccopy}(e') \). We have \( e \preceq_{u|A+B} e' : e \preceq_{u+B+B'} e' : \text{ccopy}(e) \preceq_{u|B+B'} \text{ccopy}(e') \) [since \( \text{ccopy}(e) \preceq_{u|B+B'} e \) and \( e' \preceq_{u|B+B'} \text{ccopy}(e') \)] : \( \text{ccopy}(e) \preceq_{u} \text{ccopy}(e') : \phi(e) \preceq_{u} \phi(e') \).

- If \( e \in E_u|A \) and \( e' \in E_u|B \), then \( \phi(e) = e \) and \( \phi(e') = \text{ccopy}(e') \). We have \( e \preceq_{u|A+B} e' : e \preceq_{u} e' : e \preceq_{u} \text{ccopy}(e') \) [since \( e' \preceq_{u|B+B'} \text{ccopy}(e') \)] : \( e \preceq_{u|A+B'} \text{ccopy}(e') : \phi(e) \preceq_{u|A+B'} \phi(e') \).

- If \( e \in E_u|B \) and \( e' \in E_u|A \), then \( \phi(e) = \text{ccopy}(e) \) and \( \phi(e') = e' \). We have \( e \preceq_{u|A+B} e' : e \preceq_{u} e' : \text{ccopy}(e) \preceq_{u|B+B'} \text{ccopy}(e') \) : \( \text{ccopy}(e) \preceq_{u|A+B'} \text{ccopy}(e') : \phi(e) \preceq_{u|A+B'} \phi(e') \).

\[ \blacksquare \]

**Corollary 5.45.** Let \( \sigma : A \rightarrow B \) be an asynchronous process and \( \text{cc}_B : B \rightarrow B' \) be the alternating copycat over \( B \). If \( u \in \sigma \downarrow \text{cc}_B \) and \( u \uparrow B + B' \) has even length, then \( u \uparrow A + B' \preceq u \uparrow A + B \).

**Proof.** Let \( u \in \sigma \downarrow \text{cc}_B \) and \( u \uparrow B + B' \) has even length. Using Lemma 5.44, we find \( u \uparrow A + B' \preceq^0 u \uparrow A + B \). We then use Lemma 5.33 to conclude that \( u \uparrow A + B' \preceq u \uparrow A + B \). \[ \blacksquare \]

We need the following ancillary notions in the sequel.

**Definition 5.46.** Define the preorder \( \preceq_{int} \) on \( \text{int}_A(A,B,C) \) as the relation satisfying \( u' \preceq_{int} u \) when \( u \uparrow B = u' \uparrow B \) and there is a bijection \( \phi : E_u \rightarrow E_{u'} \) such that the following conditions hold.

1. \( \lambda_u = \lambda_{u'} \circ \phi \)
2. \( (\forall e, e' \in E_u)(e \prec_u e' \text{ if and only if } \phi(e) \prec_u \phi(e')) \)
3. If \( e \preceq_u e' \) and \([e \text{ is a C-input or a B-input}]\) and \([e' \text{ is a C-output or a B-output}]\), then \( \phi(e) \preceq_{u'} \phi(e') \)
4. If \( e \preceq_u e' \) and \([e \text{ is a B-output or an A-output}]\) and \([e' \text{ is a B-input or an A-input}]\), then \( \phi(e) \preceq_{u'} \phi(e') \)
5.2. ASYNCHRONOUS PROCESSES

**Definition 5.47.** Define the preorder $\preceq^{\text{int}}$ on $\text{int}_A(A, B, C)$ as the least reflexive and transitive relation satisfying for asynchronous traces $u$ and $u'$, $u \preceq^{\text{int}} u'$ when $u$ is a prefix of $u'$ and $[u'] = [u] \cdot e$, 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 $\text{int}_A(A, B, C)$ 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 $u \preceq^{\text{int}} u'$, then $u_0 \preceq u'_0$.

**Lemma 5.49.** Let $u$ and $u'$ be interaction traces in $\text{int}_A(A, B, C)$ 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 $u \preceq^{\text{int}} u'$, then $u_0 \preceq u'_0$.

**Lemma 5.50.** Let $u$ and $u'$ be interaction traces in $\text{int}_A(A, B, C)$. If $u \preceq u'$, then $u \upharpoonright A + C \preceq u' \upharpoonright A + C$.

**Lemma 5.51.** The asynchronous identity is an identity to asynchronous processes.

*Proof.* Let $\sigma : A \rightarrow B$ be an asynchronous process and $id_B : B \rightarrow B'$.

We first prove $\sigma \subseteq \sigma; id_B$. Let $s \in \sigma$. We will find an interaction trace $u \in \text{int}_A(A, B, B')$ 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.

- $E_u = E_s|A + E_s|B + E_s|B$. Let us refer to these subsets using injections $\text{in}_1 : E_s|A \rightarrow E_u$, $\text{in}_2 : E_s|B \rightarrow E_u$ and $\text{in}_3 : E_s|B \rightarrow E_u$.
- $\lambda_u = \lambda_s|A + \lambda_s|B + \lambda_s|B$
- The preorder $\preceq_u$ is defined as follows. For all $e, e' \in E_s$, $e \preceq u e' \iff \text{in}_1(e) \preceq_u \text{in}_1(e')$ 
- P1 if $e, e' \in E_s|A$, then $e \preceq s e' \iff \text{in}_1(e) \preceq_u \text{in}_1(e')$
- P2 if $e \in E_s|A$ and $e' \in E_s|B$, then $[e \preceq s e' \iff \text{in}_1(e) \preceq_u \text{in}_2(e') \iff \text{in}_1(e) \preceq_u \text{in}_3(e')]$
- P3 if $e \in E_s|B$ and $e' \in E_s|A$, then $[e \preceq s e' \iff \text{in}_2(e) \preceq_u \text{in}_1(e') \iff \text{in}_3(e) \preceq_u \text{in}_1(e')]$
- P4 if $e, e' \in E_s|B$ and $e \neq e'$, then $[e \preceq s e' \iff \text{in}_2(e) \preceq_u \text{in}_2(e') \iff \text{in}_3(e) \preceq_u \text{in}_3(e') \iff \text{in}_2(e) \preceq_u \text{in}_3(e') \iff \text{in}_3(e) \preceq_u \text{in}_2(e')]$
---

**P5** \( e \in E_{s|B} \Leftrightarrow \text{in}_2(e) \preceq_u \text{in}_2(e) \Leftrightarrow \text{in}_3(e) \preceq_u \text{in}_3(e) \)

**P6** \( e \in E_{s|B} \) and \( \pi \circ \lambda_s(e) = i \Leftrightarrow \text{in}_3(e) \preceq_u \text{in}_2(e) \)

**P7** \( e \in E_{s|B} \) and \( \pi \circ \lambda_s(e) = o \Leftrightarrow \text{in}_2(e) \preceq_u \text{in}_3(e) \)

- The function \( \odot_u \) is defined as follows. For all \( e, e' \in E_s \),
  
  **J1** if \( e, e' \in E_{s|A} \), then \( e \odot_u e' \Leftrightarrow \text{in}_1(e) \odot_u \text{in}_1(e') \)
  
  **J2** if \( e \in E_{s|B} \) and \( e' \in E_{s|A} \), then \( e \odot_u e' \Leftrightarrow \text{in}_2(e) \odot_u \text{in}_1(e') \)
  
  **J3** if \( e, e' \in E_{s|B} \), then \( e \odot_u e' \Leftrightarrow \text{in}_2(e) \odot_u \text{in}_2(e') \Leftrightarrow \text{in}_3(e) \odot_u \text{in}_3(e') \)
  
  **J4** \( \lambda_s(e) \in I_B \Leftrightarrow \text{in}_3(e) \odot_u \text{in}_2(e) \)
  
  **J5** The only pointers in \( \odot_u \) are those specified by (J1)–(J4); that is,
  
  - if \( e \in E_{s|A} \) and \( e' \in E_{s|B} \), then \( [\text{in}_1(e) \not\prec_u \text{in}_2(e') \text{ and } \text{in}_1(e) \not\prec_u \text{in}_3(e')] \),
  
  - if \( e \in E_{s|B} \), then \( \text{in}_2(e) \not\prec_u \text{in}_3(e') \),
  
  - if \( e \in E_{s|B} \) and \( e' \in E_{s|A} \), then \( \text{in}_3(e) \not\prec_u \text{in}_1(e') \).

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 \( \text{in}_j(e_1) \preceq_u \text{in}_k(e_2) \) and \( \text{in}_k(e_2) \preceq_u \text{in}_l(e_3) \) 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 \( \text{in}_j(e_1) \preceq_u \text{in}_l(e_3) \).

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 \( e' \preceq_u e \) \( e \Rightarrow e = e' \). 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), \( e \preceq_u e' \Leftrightarrow \text{in}_1^{-1}(e) \preceq_s \text{in}_1^{-1}(e') \) and \( e' \preceq_u e \Leftrightarrow \text{in}_1^{-1}(e') \preceq_s \text{in}_1^{-1}(e) \). Since \( \preceq_s \) is antisymmetric, we conclude that \( \text{in}_1^{-1}(e) = \text{in}_1^{-1}(e') \) and so \( e = e' \).
5.2. ASYNCHRONOUS PROCESSES

2. $\lambda(e) \in L_A$ and $\lambda(e') \in L_B$. By (P2), $e \preceq u e' \iff in_1^{-1}(e) \preceq_s in_2^{-1}(e')$ and by (P3) $e' \preceq_u e \iff in_2^{-1}(e') \preceq_s in_1^{-1}(e)$. 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), $e \preceq_u e' \iff in_2^{-1}(e) \preceq_s in_3^{-1}(e')$ and $e' \preceq_u e \iff in_3^{-1}(e') \preceq_s in_2^{-1}(e)$. 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.
   - $in_2^{-1}(e) \neq in_1^{-1}(e')$. By (P4), $e \preceq_u e' \iff in_2^{-1}(e) \preceq_s in_1^{-1}(e')$ and $e' \preceq_u e \iff in_1^{-1}(e') \preceq_s in_2^{-1}(e)$. Since $\preceq_s$ is antisymmetric, we conclude that $in_1^{-1}(e) = in_2^{-1}(e')$. This is a contradiction since $in_2^{-1}(e) \neq in_1^{-1}(e')$.
   - $in_1^{-1}(e) = in_2^{-1}(e')$. 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 $e' \preceq_u e$.

- 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 $\cap_u$ is a function, i.e. no event is assigned more than one cause.

Now, we show $(\forall e, e' \in E_u)(e \cap_u e' \Rightarrow (e \leq_u e \text{ and } \lambda_u(e) \vdash \lambda_u(e'))$. Suppose $e \cap_u e'$. By definition of $\cap_u$, we have the following cases.

- If $e, e' \in E_u|A$, then $e \cap_u e' : \L r 1^{-1}(e) \L r \L r 1^{-1}(e')$ by (J1) : $\L r 1^{-1}(e) \leq_s \L r 1^{-1}(e') : e \leq_u e'$ by (P1)
- If $e, e' \in E_u|B$, then $e \cap_u e' : \L r 2^{-1}(e) \L r \L r 2^{-1}(e')$ by (J3) : $\L r 2^{-1}(e) \leq_s \L r 2^{-1}(e') : e \leq_u e'$ by (P4)
- If $e, e' \in E_u|B'$, then $e \cap_u e' : \L r 3^{-1}(e) \L r \L r 3^{-1}(e')$ by (J3) : $\L r 3^{-1}(e) \leq_s \L r 3^{-1}(e') : e \leq_u e'$ by (P4)
- If $e \in E_u|B$ and $e' \in E_u|B$, then $e \cap_u e' : \L r 3^{-1}(e) = \L r 2^{-1}(e)$ and $\lambda_s(\L r 3^{-1}(e)) \in I_B$ by (J4) : $\lambda_s(\L r 3^{-1}(e)) = i : e \leq_u e'$ by (P6)
- If $e \in E_u|B$ and $e' \in E_u|A$, then $e \cap_u e' : \L r 2^{-1}(e) \L r \L r 1^{-1}(e)$ by (J2) : $\L r 2^{-1}(e) \leq_s \L r 1^{-1}(e) : e \leq_u e'$ 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. $u \uparrow B + B' \in id_B$ and $u \uparrow A + B = s$ and $u \uparrow A + B' = s$. For $u \uparrow B + B' \in id_B$, we show that $u \uparrow B + B' \in cc_B$. Let

$$\text{iccopy}_{u|B + B'}(e) = \begin{cases} 
\L r 3^{-1}(e) & \text{if } \lambda_u(e) \in \text{lin}_2(B) \\
\L r 2^{-1}(e) & \text{if } \lambda_u(e) \in \text{lin}_3(B)
\end{cases}$$

We verify that $u \uparrow B + B'$ with $\text{iccopy}_{u|B + B'}$ satisfies the conditions of Definition 5.36. Let $e$ be an output in $u \uparrow B + B'$.

- We first show that $\text{iccopy}_{u|B + B'}(e) \leq_{u|B + B'} e$. If $e$ is a $B'$-event, then $\L r 3^{-1}(e) \in E_s$ and $\text{iccopy}_{u|B + B'}(e) = \L r 2^{-1}(e)$. Note that if $e$ is a $B'$-output in $u \uparrow B + B'$ then $\L r 3^{-1}(e)$ is an output in $s$. By (P7), we have $\L r 2^{-1}(e) \leq_s \L r 2^{-1}(e)$. So, $\text{iccopy}_{u|B + B'}(e) \leq_u e$ and hence, $\text{iccopy}_{u|B + B'}(e) \leq_{u|B + B'} e$. If $e$ is a $B$-event, then $\L r 2^{-1}(e) \in E_s$ and $\text{iccopy}_{u|B + B'}(e) = \L r 2^{-1}(e)$.
5.2. ASYNCHRONOUS PROCESSES

in \(3\)(in\(_2^{-1}(e)\)). Note that if \(e\) is a \(B\)-output in \(u \upharpoonright B + B'\) then in\(_2^{-1}(e)\) is an input in \(s\).

By (P6), we have in\(_3\)(in\(_2^{-1}(e)\)) \(\leq_u\) in\(_2\)(in\(_2^{-1}(e)\)). So, iccopy\(_{u|B+B'}(e)\) \(\leq_u\) \(e\) and hence, iccopy\(_{u|B+B'}(e)\) \(\leq_{u|B+B'} e\).

- We now show that there is no \(e' \in E_{u|B+B'}\) such that \(e' \neq e\) and \(e' \neq\) iccopy\(_{u|B+B'}(e)\) and iccopy\(_{u|B+B'}(e)\) \(\leq_{u|B+B'} e'\). Suppose to the contrary that there is \(e' \in E_{u|B+B'}\) such that \(e' \neq e\) and \(e' \neq\) iccopy\(_{u|B+B'}(e)\) and iccopy\(_{u|B+B'}(e)\) \(\leq_{u|B+B'} e\). So, iccopy\(_{u|B+B'}(e)\) \(\leq_u\) \(e'\) \(\leq_u\) \(e\). We have the following cases.

  - If \(e\) is a \(B'\)-event, then in\(_3^{-1}(e)\) \(\in E_s\) and iccopy\(_{u|B+B'}(e)\) = in\(_2\)(in\(_3^{-1}(e)\)).

    * If \(e'\) is a \(B\)-event, then using (P4), we have iccopy\(_{u|B+B'}(e)\) \(\leq_u\) \(e'\) \(\Rightarrow\) in\(_2\)(in\(_3^{-1}(e)\)) \(\leq_u\) \(e'\) \(\Rightarrow\) in\(_2\)(in\(_3^{-1}(e)\)) \(\leq_s\) in\(_2\)(in\(_3^{-1}(e)\)) \(\leq_s\) in\(_3\)(in\(_3^{-1}(e)\)) \(\leq_u\) \(e'\) \(\Rightarrow\) in\(_3\)(in\(_3^{-1}(e)\)) \(\leq_u\) \(e'\) \(\Rightarrow\) in\(_3\)(in\(_3^{-1}(e)\)) \(\leq_u\) in\(_3\)(in\(_3^{-1}(e)\)) \(\Rightarrow\) e \(\leq_u\) \(e'\) \(\Rightarrow\) \(e'\) \(\leq_u\) e. Since \(\leq_s\) is antisymmetric, we get in\(_3^{-1}(e)\) = in\(_3^{-1}(e)\). We use (P5) to find that in\(_2\)(in\(_3^{-1}(e)\)) \(\leq_u\) in\(_2\)(in\(_3^{-1}(e)\)) \(\Rightarrow\) iccopy\(_{u|B+B'}(e)\) \(\leq_u\) \(e'\) \(\Rightarrow\) \(e'\) \(\leq_u\) iccopy\(_{u|B+B'}(e)\) \(\Rightarrow\) in\(_3\)(in\(_3^{-1}(e)\)) \(\leq_u\) in\(_3\)(in\(_3^{-1}(e)\)) \(\Rightarrow\) e \(\leq_u\) \(e'\) \(\Rightarrow\) e \(\leq_u\) e. We then use the fact that \(\leq_u\) is antisymmetric to find that e = \(e'\) = iccopy\(_{u|B+B'}(e)\) which contradicts our assumptions.

    * If \(e'\) is a \(B'\)-event, then using (P4), we have iccopy\(_{u|B+B'}(e)\) \(\leq_u\) \(e'\) \(\Rightarrow\) in\(_2\)(in\(_3^{-1}(e)\)) \(\leq_u\) \(e'\) \(\Rightarrow\) in\(_2\)(in\(_3^{-1}(e)\)) \(\leq_s\) in\(_3\)(in\(_3^{-1}(e)\)) \(\leq_s\) in\(_3\)(in\(_3^{-1}(e)\)) \(\leq_u\) in\(_3\)(in\(_3^{-1}(e)\)) \(\Rightarrow\) e \(\leq_u\) \(e'\) \(\Rightarrow\) \(e'\) \(\leq_u\) e. Since \(\leq_s\) 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 \(u \upharpoonright B + B'\) with iccopy\(_{u|B+B'}\) satisfies condition (C2) of Definition 5.36 is clear from the definition of \(\lambda_u\).

- We verify that condition (C3) of Definition 5.36 holds.

  - Suppose \(\lambda_{u|B+B'}(e) \in I_B\) or \(\lambda_{u|B+B'}(iccopy_{u|B+B'}(e)) \in I_B'\). It follows that \(\lambda_s\)(in\(_2^{-1}(e)\)) \(\in\) I\(_B\). By (J4), in\(_3\)(in\(_2^{-1}(e)\)) \(\sim_u\) in\(_2\)(in\(_2^{-1}(e)\)). Therefore, iccopy\(_{u|B+B'}(e)\) \(\sim_u\) e. Hence, iccopy\(_{u|B+B'}(e)\) \(\sim_u\) \(e\).

  - Suppose \(\lambda_{u|B+B'}(e) \notin I_B\) and \(\lambda_{u|B+B'}(iccopy_{u|B+B'}(e)) \notin I_B'\). Let \(e\) be a \(B\)-event.

    By definition of \(\lambda_u\), we have \(\lambda_s\)(in\(_2^{-1}(e)\)) \(\notin I_B\). Since \(s\) is justified, there is \(e' \in E_s\)
such that \( e' \sim u \) \( \text{in}^{-1}_2(e) \). Using (J3), we get \( \text{in}_2(e') \sim_u \text{in}_2(\text{in}^{-1}_2(e)) \therefore \text{in}_2(e') \sim_u e \therefore \text{in}_2(e') \sim_u [B + B'] e \) and \( \text{in}_3(e') \sim_u \text{in}_3(\text{in}^{-1}_2(e)) \therefore \text{in}_3(e') \sim_u \text{iccopy}_{u|B + B'}(e) \therefore \text{in}_3(e') \sim_u [B + B'] \text{iccopy}_{u|B + B'}(e) \). Since \( \text{in}_3(e') \) is an \( B' \)-output in \( u \upharpoonright B + B' \), we get \( \text{iccopy}_{u|B + B'}(\text{in}_3(e')) = \text{in}_2(\text{in}^{-1}_3(\text{in}_3(e'))) = \text{in}_2(e') \). The case where \( e \) is a \( B' \)-event is similar.

We briefly argue that \( u \upharpoonright A + B = s \). We have \( E_{u|A+B} = E_{s|A} + E_{s|B} \). There is a bijection \( \phi : E_s \to E_{u|A+B} \) defined as

\[
\phi(e) = \begin{cases} 
\text{in}_1(e) & \text{if } \lambda_s(e) \in \text{inl}(L_A) \\
\text{in}_2(e) & \text{if } \lambda_s(e) \in \text{inr}(L_B) 
\end{cases}
\]

We need to prove that for all \( e, e' \in E_s, e \preceq_s e' \iff \phi(e) \preceq_{u|A+B} \phi(e') \). We have \( e \preceq_s e' \iff \text{in}_j(e) \preceq_u \text{in}_k(e') \iff \text{in}_j(e) \preceq_{u|A+B} \text{in}_k(e') \) where \( j, k \in \{1, 2\} \).

The proof that \( u \upharpoonright A + B' = s \) is similar.

Next, we show the opposite inclusion, i.e. \( \sigma; id_B \subseteq \sigma \). It is sufficient to prove that for any \( u \in \text{int}_A(A, B, B') \)

if \( u \upharpoonright A + B = s \in \sigma \) \hspace{1cm} (5.1)

and \( u \upharpoonright B + B' = t \in id_B \) \hspace{1cm} (5.2)

then \( u \upharpoonright A + B' \in \sigma \) \hspace{1cm} (5.3)

We will show that \( u \upharpoonright A + B' \) 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 \( E_u \) such that \( \text{o copy}_w(e) \) 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,

- \( E_{u_1} = E_u \).
- \( \lambda_{u_1} = \lambda_u \).
5.2. ASYNCHRONOUS PROCESSES

- \( \sim_{u_1} = \cap_u \).

- Define \( P' = \{ e \in E_u \mid \lambda_u(e) \in L_{B'} \text{ and } \pi \circ \lambda_I(e) = i \text{ and ocopy}_I(e) \text{ is undefined} \} \).
  Let \( \leq_{u_1} \) be defined by the following, for all \( e, e' \in E_u \).

1. If \( e, e' \in E_u \setminus P' \), then \( e \leq_{u_1} e' \iff e \leq_{u_1} e' \).

2. If \( e, e' \in P' \), then \( e \leq_{u_1} e' \iff e \leq_{u_1} e' \).

3. If \( e \in E_u \setminus P' \) and \( e' \in P' \), then \( e \leq_{u_1} e' \).

We can show that \( u_1 \) is an asynchronous trace; the proof is omitted. We show that \( u \leq^{\text{int}} u_1 \). First note that \( u \upharpoonright B = u_1 \upharpoonright B \). Define the bijection \( \phi : E_{u_1} \rightarrow E_u \) as the identity. We have by definition of \( u_1 \) that \( \lambda_{u_1} = \lambda_u \) and \( \cap_{u_1} = \cap_u \). For the last condition of Definition 5.46 let \( e \leq_{u_1} e' \).

- If \([e] \) is a \( B' \)-input or a \( B \)-input \) and \([e'] \) is a \( B' \)-output or a \( B \)-output, \( e, e' \in E_{u_1} \setminus P' \) because \( e' \) is not a \( B' \)-input and if \( e \in P' \), then \( e \leq_{u_1} e' \) implies \( e' \in P' \) by (P1.2). Using (P1.1), we find that \( e \leq_{u} e' \).

- If \([e] \) is a \( B \)-output or an \( A \)-output \) and \([e'] \) is a \( B \)-input or an \( A \)-input, \( e, e' \in E_{u_1} \setminus P' \). Using (P1.1), we find that \( e \leq_{u} e' \).

Note that \( u_1 \upharpoonright A + B = s \). Applying Lemma 5.48 to \( u \leq^{\text{int}} u_1 \) we get \( u \upharpoonright A + B' \leq u_1 \upharpoonright A + B' \). Let us call this result (\( */ \)).

2. Let \( u_2 \) be a trace that is like \( u_1 \), but where all pending \( B' \)-inputs are removed. It is clear that \( u_1 \leq^{\text{int}} u_2 \) and \( u_2 \upharpoonright A + B = s \). Applying Lemma 5.49 to \( u_1 \leq^{\text{int}} u_2 \) we get \( u_1 \upharpoonright A + B' \leq u_2 \upharpoonright A + B' \). Let us call this result (\( \mathcal{A} \)).

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 \( P = \{ e \in E_{u_2} \mid \lambda_{u_2}(e) \in L_B \text{ and } \pi \circ \lambda_{u_2|B+B'}(e) = i \text{ and ocopy}_{u_2|B+B'}(e) \text{ is undefined} \} \) and

- \( E_{u_3} = E_{u_2} + P \).
- \( \lambda_{u_3} = \lambda_{u_2} + \lambda_{u_2} \upharpoonright P \).
- Let \( \leq_{u_3} \) be defined by the following, for all \( e, e' \in E_{u_2} \).
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,

- $E_{u_4} = E_{u_3}$.
- $\lambda_{u_4} = \lambda_{u_3}$.
- $\smallsetminus_{u_4} = \smallsetminus_{u_3}$.

Define $\text{copy} = \text{copy}_{u_3|B+B'} \cup \text{copy}_{u_3|B+B'}$. We have copy is a well defined injection because the domains of $\text{copy}_{u_3|B+B'}$ and $\text{copy}_{u_3|B+B'}$ are disjoint, and so are their codomains. Note that copy is an involution. The preorder $\preceq_{u_4}$ is defined as follows.

For all $e, e' \in E_{u_4}$,

- $P_{4.1}$ if $e, e' \in E_{u_3|A}$, then $e \preceq_{u_3} e' \iff e \preceq_{u_4} e'$
- $P_{4.2}$ if $e \in E_{u_3|A}$ and $e' \in E_{u_3|B}$, then $[e \preceq_{u_3} e' \iff e \preceq_{u_4} e' \iff e \preceq_{u_4} \text{copy}(e')]$
- $P_{4.3}$ if $e \in E_{u_3|B}$ and $e' \in E_{u_3|A}$, then $[e \preceq_{u_3} e' \iff e \preceq_{u_4} e' \iff \text{copy}(e) \preceq_{u_4} e']$
- $P_{4.4}$ if $e, e' \in E_{u_3|B}$ and $e \neq e'$, then $[e \preceq_{u_3} e' \iff e \preceq_{u_4} e' \iff \text{copy}(e) \preceq_{u_4} \text{copy}(e')]$
- $P_{4.5}$ if $e \in E_{u_3|B}$ and $\pi \circ \lambda_{u_3|B+B'}(e) = i \iff e \preceq_{u_4} \text{copy}(e)$
- $P_{4.6}$ if $e \in E_{u_3|B}$ and $\pi \circ \lambda_{u_3|B+B'}(e) = o \iff \text{copy}(e) \preceq_{u_4} e$.

We show that $(\forall e, e' \in E_{u_4})(e \smallsetminus_{u_4} e' \Rightarrow e \preceq_{u_4} e')$ via a case study on the labels of $e, e'$. 

The proof that $u_3$ is an asynchronous trace is omitted. It is clear that $u_2 \preccurlyeq u_3$ and $u_3 \preccurlyeq A + B = s$. Note that $u_3 \mid B + B'$ 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 $u_2 \mid A + B' \preccurlyeq u_3 \mid A + B'$. Let us call this result (•).
5.2. ASYNCHRONOUS PROCESSES

- If \( e, e' \in E_{u_4 \setminus A} \), then \( e \cdot e_4 \cdot e' \cdot e' \cdot e' \cdot e' \cdot e' \cdot e' \) as \( u_3 \) is well-formed. \( e \cdot e \cdot e' \) by (P4.1).

- If \( e, e' \in E_{u_4 \setminus B} \), then \( e \cdot e_4 \cdot e' \cdot e' \cdot e' \cdot e' \cdot e' \) as \( u_3 \) is well-formed. \( e \cdot e \cdot e' \) by (P4.4).

- \( e, e' \in E_{u_4 \setminus B'} \), then \( e \cdot e_4 \cdot e' \cdot e' \cdot e' \cdot e' \). By Definition 5.37, \( \text{copy}(e) \cdot e_4 \cdot e' \cdot e' \). \( \text{copy}(e') \leq u_3 \cdot \text{copy}(e') \). By (P4.4), \( \text{copy}(\text{copy}(e)) \leq u_4 \cdot \text{copy}(\text{copy}(e')) \). So, \( e \leq u_4 \cdot e' \) since \( \text{copy} \) is involutary.

- \( e \in E_{u_4 \setminus B} \) and \( e' \in E_{u_4 \setminus A} \), then \( e \cdot e_4 \cdot e' \cdot e' \cdot e' \cdot e' \) as \( u_3 \) is well-formed. \( e \cdot e \cdot e' \) by (P4.3).

- \( e \in E_{u_4 \setminus B'} \) and \( e' \in E_{u_4 \setminus B} \), then \( e \cdot e_4 \cdot e' \cdot e' \cdot e' \). By Definition 5.37, \( e = \text{icopy}_{u_3 \setminus B+B'}(e') \). We then use (P4.6) to find that \( e \leq u_4 \cdot e' \).

- All other label assignments are impossible.

The rest of the proof that \( u_4 \) is an asynchronous trace is omitted. We show that \( u_3 \leq u_4 \).

Note that \( u_3 \uparrow B = u_4 \uparrow B \). Define the bijection \( \phi : E_{u_4} \rightarrow E_{u_3} \) as the identity. We have by definition of \( u_4 \) that \( \lambda_{u_4} = \lambda_{u_3} \) and \( \cdot u_4 = \cdot u_3 \). For the last condition of Definition 5.46, let \( e \leq u_4 \cdot e' \).

- If \( e \) is a \( B' \)-input and \( e' \) is a \( B' \)-output, then by (P4.4), \( \text{copy}^{-1}(e) \leq u_3 \cdot \text{copy}^{-1}(e') \).

Since \( \text{copy} \) is involutary, \( \text{copy}(e) \leq u_3 \cdot \text{copy}(e') \). Since \( u_3 \uparrow B + B' \in \text{id}_B \), we know that \( \text{icopy}_{u_3 \setminus B+B'}(\text{copy}(e)) \leq u_3 \cdot \text{copy}(e) \) and \( \text{copy}(e') \leq u_3 \cdot \text{copy}^{-1}(\text{copy}(e')) \).

So, \( e \leq u_3 \cdot e' \).

- If \( e \) is a \( B' \)-input and \( e' \) is a \( B \)-output, then by (P4.4), \( \text{copy}(e) \leq u_3 \cdot e' \). We use the fact that \( \text{icopy}_{u_3 \setminus B+B'}(\text{copy}(e)) \leq u_3 \cdot \text{copy}(e) \) to find that \( e \leq u_3 \cdot e' \).

- If \( e \) is a \( B \)-input and \( e' \) is a \( B' \)-output, then by (P4.4), \( e \leq u_3 \cdot \text{copy}(e') \). We use the fact that \( \text{copy}(e') \leq u_3 \cdot \text{copy}^{-1}(\text{copy}(e')) \) to find that \( e \leq u_3 \cdot e' \).

- If \( e \) is a \( B \)-input and \( e' \) is a \( B \)-output, then by (P4.4), \( e \leq u_3 \cdot e' \).

- If \( e \) is a \( B \)-output or an \( A \)-output, then \( e' \) 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 \( u_3 \preceq^\text{int} u_4 \) we get \( u_3 \upharpoonright A + B' \preceq u_4 \upharpoonright A + B' \). Let us call this result (♦). 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 \( u_4 \upharpoonright A + B' \preceq s \). Let us call this result (▼).

Putting together (⋆), (▲), (●), (♦) and (▼) yields

\[
\begin{align*}
\upharpoonright u_1 &\preceq u_1 \upharpoonright A + B' \preceq u_2 \upharpoonright A + B' \preceq u_3 \upharpoonright A + B' \preceq u_4 \upharpoonright A + B' \preceq s
\end{align*}
\]

Hence, \( u \upharpoonright A + B' \in \sigma \).

The proof that \( \text{id}_A ; \sigma = \sigma \) is similar.

\[\blacksquare\]

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.

\[\blacksquare\]

Lemma 5.53. Tensor preserves the identity, i.e., \( \text{id}_A \otimes \text{id}_B = \text{id}_{A \otimes B} \).

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 \( \text{SynProc}_P \). 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 \( (E_s, \lambda_s, \mathbin{\bowtie}_s) \) and \( (E_t, \lambda_t, \mathbin{\bowtie}_t) \) are \( \phi \)-isomorphic, i.e. \( \lambda_s = \lambda_t \circ \phi \) and \( \mathbin{\bowtie}_s = \mathbin{\bowtie}_t \circ \phi \), and \( \phi \) 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.
5.3. ROUND ABSTRACTION ON CAUSAL PROCESSES

**Definition 5.55 (Partial round abstraction).** For causal processes $\sigma$ and $\tau$ over $A$, we say that $\tau$ is a partial round abstraction of $\sigma$, 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 $\sigma, \sigma' : A \rightarrow B$ and $\tau, \tau' : B \rightarrow C$ be causal processes such that $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$. For all $u' \in \sigma' \vec{j} \tau'$, there are interaction traces $u, v \in \text{int}_I(A,B,C)$ and traces $s \in \sigma$ and $t \in \tau$ satisfying,

1. $u \sqsubseteq u' \text{ and } u \restriction A + B = s \text{ and } u \restriction B \in \Pi(t \restriction B) \text{ and } u \restriction C = t \restriction C$
2. $v \sqsubseteq u' \text{ and } v \restriction B + C = t \text{ and } v \restriction B \in \Pi(s \restriction B) \text{ and } v \restriction A = s \restriction A$

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 $u' \in \sigma' \vec{j} \tau'$; that is, $u' \restriction A + B \in \sigma'$ and $u' \restriction B + C \in \tau'$. Since $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$, it follows there are $s \in \sigma$ and $t \in \tau$ such that $s \sqsubseteq u' \restriction A + B$ and $t \sqsubseteq u' \restriction B + C$. So, let $\phi : E_s \rightarrow E_{u'\restriction A + B}$ and $\psi : E_t \rightarrow E_{u'\restriction B + C}$ be bijections such that $\lambda_s = \lambda_{u'\restriction A + B} \circ \phi$ and $\lambda_t = \lambda_{u'\restriction B + C} \circ \psi$. Without loss of generality, we will assume that $\phi$ and $\psi$ are identities, i.e. that $E_s = E_{u'\restriction A + B}$ and $E_t = E_{u'\restriction B + C}$. It follows that $E_s \restriction B = E_t \restriction B$ and so $s \restriction B$ is a permutation of $t \restriction B$.

The interaction trace $u$ is defined as follows.

- $E_u = E_{u'}$
- $\lambda_u = \lambda_{u'}$
- $\leadsto_u = \leadsto_{u'}$

- 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 $e, e' \in E_u$.

  \[ P1 \ e \prec_{u'} e' \iff e \preceq_1 e'. \]
P2 If \( e, e' \in E_s \), then \( e \preceq_i e' \Leftrightarrow e \preceq_2 e' \).

P3 If \( e, e' \in E_{i|C} \), then \( e \preceq_t e' \Leftrightarrow e \preceq_2 e' \).

P4 If \( e \in E_{i|C} \) and \( e' \in E_{i|B} \), then \( e \approx_{u'} e' \Leftrightarrow e \preceq_2 e' \).

P5 If \( e \in E_{i|C} \) and \( e' \in E_{i|A} \), then \( e \approx_{u'} e' \Leftrightarrow e \preceq_2 e' \).

P6 If \( e \in E_s \) and \( e' \in E_{i|C} \), 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 \( \bowtie_u \) because \( u' \) is justified and \( E_u = E_{u'} \) and \( \bowtie_u = \bowtie_{u'} \). We still need to verify that \( (\forall e, e' \in E_u)(e \bowtie_u e' \Rightarrow e \preceq_u e') \). Suppose \( e \bowtie_u e' \). Then, \( e \bowtie_u e' \Leftrightarrow e \bowtie_{u'} e' \Leftrightarrow e \bowceq_{u'} e' \). There are two cases.

- \( e \bowceq_{u'} e' \). By (P1), \( e \bowceq_u e' \).

- \( e \bowceq_{u'} e' \). 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 \( e \bowtie_{u'} e' \Leftrightarrow e \bowtie_{u'} \bowtie_{B+C} e' \Leftrightarrow e \bowtie_t e' \) because \( t \subseteq u' \mid B + C \). 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 \( e \bowtie_{u'} e' \Leftrightarrow e \bowtie_{u'} \bowtie_{A+B} e' \Leftrightarrow e \bowtie_s e' \) because \( s \subseteq u' \mid A + B \). 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: \( \bowtie_u = \bowtie_{u'} \) and \( \bowtie_{u|A+B} = \bowtie_s \) and \( \bowtie_{u|C} = \bowtie_{i|C} \).

We can now restate the main theorems.

**Theorem 5.57** (Soundness I). For any causal processes \( \sigma, \sigma' : A \rightarrow B \) and \( \tau, \tau' : B \rightarrow C \), if \( \sigma \preceq \sigma' \) and \( \tau \preceq \tau' \) and \( \sigma \bowtie \tau \), then \( \sigma; \tau \preceq \sigma'; \tau' \).

**Theorem 5.58** (Soundness II). For any causal processes \( \sigma, \sigma' : A \rightarrow B \) and \( \tau, \tau' : B \rightarrow C \), if \( \sigma \preceq \sigma' \) and \( \tau \preceq \tau' \) and \( \sigma' \bowtie \tau' \), then \( \sigma; \tau \preceq \sigma'; \tau' \).
5.3. ROUND ABSTRACTION ON CAUSAL PROCESSES

The proofs as exactly as given in Chapter [4]

**Theorem 5.59** (Soundness III). For any causal processes $\sigma, \sigma' : A \rightarrow B$ and $\tau, \tau' : B \rightarrow C$, 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 $\sqcap_{\nu}$, which we define as $\sqcap_s + \sqcap_t$. The proof that $v$ is justified and pointers are preserved between $v$ and $v'$ is omitted.

The above theorems also hold if $\sigma$ and $\tau$ are asynchronous and $\sigma'$ and $\tau'$ 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 $\sigma : A \rightarrow B$, $\tau : B \rightarrow C$ be asynchronous processes and $\sigma' : A \rightarrow B$, $\tau' : B \rightarrow C$ be causal processes such that $\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$. For all $u', v \in \text{int}_A(A, B, C)$ and traces $s \in \sigma$ and $t \in \tau$ satisfying,

1. $u \sqsubseteq u'$ and $u \uparrow A + B = s$ and $u \uparrow B \in \Pi_A(t \uparrow B)$ and $u \uparrow C = t \uparrow C$

2. $v \sqsubseteq u'$ and $v \uparrow B + C = t$ and $v \uparrow B \in \Pi_A(s \uparrow B)$ and $v \uparrow A = s \uparrow A$

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

The interaction trace $u$ is defined as in Lemma [5.56].

- $E_u = E_{u'}$
- $\lambda_u = \lambda_{u'}$
- $\sqcap_u = \sqcap_{u'}$
- 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 $b \neq a$. Let the relations $\preceq_1 \sqsubseteq \preceq_{u'}$ and $\preceq_2 \sqsubseteq \preceq_{u'}$ be defined as follows, for all $e, e' \in E_u$. 
CHAPTER 5. CAUSAL PROCESSES AND ASYNCHRONOUS PROCESSES

P1 \( e \prec_{w'} e' \iff e \leq_1 e' \).

P2 If \( e, e' \in E_s \), then \( e \leq_3 e' \iff e \leq_2 e' \).

P3 If \( e, e' \in E_{t|C} \), then \( e \leq_t e' \iff e \leq_2 e' \).

P4 If \( e \in E_{t|C} \) and \( e' \in E_{s|B} \), then \( e \approx_{w'} e' \iff e \leq_2 e' \).

P5 If \( e \in E_{t|C} \) and \( e' \in E_{s|A} \), then \( e \approx_{w'} e' \iff e \leq_2 e' \).

P6 If \( e \in E_s \) and \( e' \in E_{t|C} \), then \( e \not\leq_2 e' \).

We then set \( \leq_u = \leq_1 \cup \leq_2 \).

We need to check that \( \leq_u \) is antisymmetric, i.e. \( (\forall e, e' \in E_u)(e \leq_u e' \text{ and } e' \leq_u e \Rightarrow e = e') \). Let \( e \leq_u e' \) and \( e' \leq_u e \), it follows from (P1) that \( e \leq_2 e' \) and \( e' \leq_2 e \). 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 \( e \leq_2 e' \iff e \leq_3 e' \) by (P2). Similarly, \( e' \leq_2 e \iff e' \leq_3 e \) by (P2). We get \( e \leq_3 e' \) and \( e' \leq_3 e \). So, \( e = e' \) because \( \leq_3 \) 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 \( \text{SynProc}_p \), 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 \textbf{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 \cite[pp. 14–18]{GM08}—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 \textbf{SynProc}_p. Moreover, when applied to processes in \textbf{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 \cite{GS10}.

Further, our results are of particular interest to hardware synthesis using GoS \cite{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 \textit{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.


**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 follow 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 $C$ is a triple $\langle T, \eta, \mu \rangle$, where $T$ is an endofunctor in $C$, $\eta : I \rightarrow T$, $\mu : T^2 \rightarrow T$ are natural transformations, and the following diagrams commute.

\[
\begin{align*}
T^3 A \xrightarrow{\mu TA} T^2 A \\
\downarrow T_{\mu A} & \quad & \downarrow \mu_A \\
T^2 A \xrightarrow{\mu_A} TA \\
\end{align*}
\]

\[
\begin{align*}
TA \xrightarrow{\eta TA} T^2 A \\
\downarrow id_T & \quad & \downarrow id_T \\
TA \xrightarrow{\mu A} TA \\
\end{align*}
\]

Definition 6.2 (Strong monad). A strong monad on a symmetric monoidal category $C$ is a monad, $\langle T, \eta, \mu, t \rangle$ equipped with a tensorial strength natural transformation $t$ that associates to each pair of objects $A, B$, a morphism $t_{A,B} : A \otimes TB \rightarrow T(A \otimes B)$ such that the following diagrams commute.

\[
\begin{align*}
(A \otimes B) \otimes TC \xrightarrow{t_{A,B,C}} T( (A \otimes B) \otimes C) & \quad & T \otimes A \xrightarrow{t_{T,A}} T (I \otimes A) \\
\downarrow \alpha_{A,B,C} & \quad & \downarrow T \lambda_A \\
A \otimes (B \otimes TC) \xrightarrow{id_A \otimes t_{B,C}} A \otimes T( B \otimes C) \xrightarrow{t_{A,B,C}} T(A \otimes (B \otimes C)) & \quad & TA \\
\end{align*}
\]

\[
\begin{align*}
A \otimes T^2 B \xrightarrow{id_A \otimes \mu_B} A \otimes TB \xrightarrow{id_A \otimes \eta_B} A \otimes B \\
\downarrow t_{A,TB} & \quad & \downarrow \eta_{A \otimes B} \\
T( A \otimes TB) \xrightarrow{T \lambda_B} T^2 (A \otimes B) \xrightarrow{\mu_{A \otimes B}} T (A \otimes B) \\
\end{align*}
\]

The strength of a strong monad $\langle T, \eta, \mu, t \rangle$ on a symmetric monoidal category $C$ gives rise to a costrength natural transformation $t'$ that associates to each pair of objects $A$ and $B$, a morphism $t'_{A,B} : TA \otimes B \rightarrow T(A \otimes B)$ defined as follows.

\[
t'_{A,B} : TA \otimes B \xrightarrow{\gamma_{T,A,B}} B \otimes TA \xrightarrow{t_{B,A}} T(B \otimes A) \xrightarrow{T \gamma_{B,A}} T (A \otimes B)
\]

A strong monad is commutative when the following diagram commutes.

\[
\begin{align*}
T( TA \otimes B) \xrightarrow{T \lambda_B} T^2 (A \otimes B) \\
\downarrow T t'_{A,B} & \quad & \downarrow \mu_{A \otimes B} \\
TA \otimes TB \xrightarrow{T \lambda_B} T^2 (A \otimes B) \\
\end{align*}
\]

\[
\begin{align*}
T( A \otimes TB) \xrightarrow{T \lambda_B} T^2 (A \otimes B) \\
\downarrow T t'_{A,TB} & \quad & \downarrow \mu_{A \otimes B} \\
T (A \otimes TB) \xrightarrow{T \lambda_B} T^2 (A \otimes B) \\
\end{align*}
\]
**Definition 6.3** (Monoid). A monoid \( \langle M, \eta, \mu \rangle \) in a monoidal category \( C \) 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.

\[
\begin{array}{c}
\xymatrix{
(M \otimes M) \otimes M \ar[r]^{a_{M,M,M}} \ar[d]_{\mu \otimes id_M} & M \otimes (M \otimes M) \ar[r]^{id_M \otimes \mu} \ar[d]_{\mu} & M \otimes M \\
M \otimes M \ar[r]^\mu & M 
}
\end{array}
\]

If \( C \) has a symmetry \( \gamma \) 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 \( \text{SynProc}_p \) with a single label \textit{tick}. The functor \( T : \text{SynProc}_p \to \text{SynProc}_p \) is defined as follows.

- **On objects**: \( T(A) = A \otimes Ck \)
- **On morphisms**: \( T(f) = f \otimes id_{Ck} \)

We also define natural transformation (at object \( A \)) \( \eta_A : A \to T(A) \) as,

\[
\eta_A = \{ s \in \Theta_f(A) \Rightarrow A' \otimes Ck) \mid s \upharpoonright (A + A') \in id_A \}
\]

and natural transformation (at object \( A \)) \( \mu_A : T^2(A) \to T(A) \) as,

\[
\mu_A = \{ s \in \Theta_f((A \otimes Ck) \otimes Ck') \Rightarrow A' \otimes Ck') \mid s \upharpoonright (A + A') \in id_A \text{ and } s \upharpoonright (Ck + Ck') \in id_{Ck} \text{ and } s \upharpoonright (Ck' + Ck'') \in id_{Ck}\}. 
\]

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 \( \eta_A \) and \( \mu_A \) 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
ports, but $Ck$ is single-port; we show its input/output polarity with arrows for extra clarity.

The following are alternative characterisations. We have $\eta_A = \rho_A^{-1} \odot id_A \odot cs_{Ck}$ where $cs_{Ck} : I \to Ck$ is the set of traces consisting of $\text{tick}$ occurrences, i.e. $\text{tick}^*$. Similarly, $\mu_A = \alpha_{A,Ck,Ck} \odot id_A \odot sp_{Ck}$ where $sp_{Ck} : Ck \odot Ck \to Ck$ is the set of traces defined by $\{ s \in \Theta_I( Ck \otimes Ck' \Rightarrow Ck'') \mid s \upharpoonright (Ck + Ck'') \in id_{Ck} \text{ and } s \upharpoonright (Ck + Ck''') \in id_{Ck} \}$. A direct informal characterisation is $(\text{tick}, \text{tick}', \text{tick}'')^*$ where $\text{tick} \in L_{Ck}$, $\text{tick}' \in L_{Ck'}$ and $\text{tick}'' \in L_{Ck''}$.

We first show that $\langle Ck, sp_{Ck}, cs_{Ck} \rangle$ is a monoid in $\text{SynProc}_p$.

**Lemma 6.4.** The triple $\langle Ck, sp_{Ck}, cs_{Ck} \rangle$ is a monoid in $\text{SynProc}_p$.

**Proof.** We have to show the following equalities.

$$\alpha_{Ck,Ck,Ck} \odot id_{Ck} \odot sp_{Ck} \odot sp_{Ck} = sp_{Ck} \odot id_{Ck} \odot sp_{Ck}.$$  \hfill (6.1)

$$\alpha_{Ck,Ck,Ck}^{-1} \odot id_{Ck} \odot sp_{Ck} \odot sp_{Ck} = id_{Ck} \odot sp_{Ck} \odot sp_{Ck}.$$  \hfill (6.2)

$$cs_{Ck} \odot id_{Ck} \odot sp_{Ck} = \lambda_{Ck}.$$  \hfill (6.3)

$$id_{Ck} \odot cs_{Ck} \odot sp_{Ck} = cs_{Ck}.$$  \hfill (6.4)

We begin by proving (6.1). Expanding the definitions, we have,

$$LHS = \{ u \in \Theta_I( (((Ck_4 \otimes Ck_5) \otimes Ck_6) \Rightarrow (Ck_4' \otimes (Ck_5' \otimes Ck_6'))) \Rightarrow Ck_2 \otimes Ck_3) \Rightarrow Ck_1) \mid$$

$$u \upharpoonright Ck_2 + Ck_1 \in id_{Ck} \text{ and } u \upharpoonright Ck_3 + Ck_1 \in id_{Ck} \text{ and } u \upharpoonright Ck_4' + Ck_2 \in id_{Ck} \text{ and } u \upharpoonright Ck_5' + Ck_3 \in id_{Ck} \text{ and } u \upharpoonright Ck_6 + Ck_3 \in id_{Ck} \text{ and } u \upharpoonright Ck_4 + Ck_4' \in id_{Ck} \text{ and } u \upharpoonright Ck_5 + Ck_5' \in id_{Ck} \text{ and } u \upharpoonright Ck_6 + Ck_6 \in id_{Ck} \} \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_1$$

$$RHS = \{ v \in \Theta_I( (((Ck_4 \otimes Ck_5) \otimes Ck_6) \Rightarrow Ck_2 \otimes Ck_3) \Rightarrow Ck_1) \mid$$

$$v \upharpoonright Ck_2 + Ck_1 \in id_{Ck} \text{ and } v \upharpoonright Ck_3 + Ck_1 \in id_{Ck} \text{ and } v \upharpoonright Ck_4 + Ck_2 \in id_{Ck} \text{ and } v \upharpoonright Ck_4 + Ck_2 \in id_{Ck} \text{ and } v \upharpoonright Ck_4 + Ck_2 \in id_{Ck}$$
We first show that \(a_{\text{CK,CK,CK}}: \text{id}_{\text{CK}} \otimes \text{sp}_{\text{CK}} \otimes \text{sp}_{\text{CK}} \subseteq \text{sp}_{\text{CK}} \otimes \text{id}_{\text{CK}} \otimes \text{sp}_{\text{CK}}\).

Let us recall some notation introduced in Chapter 3. For \(u \in \text{id}_A\) and events \(e, e' \in E_u\), where \(u \in \text{id}_A\), let us write \(e \equiv_u e'\) when \(e \sim_u e'\) and \(\lambda_u(e) = \text{inl}(a)\) if and only if \(\lambda_u(e') = \text{inr}(a)\) and \(\lambda_u(e) = \text{inr}(a)\) if and only if \(\lambda_u(e') = \text{inl}(a)\) where \(a \in L_A\).

Let \(u \in a_{\text{CK,CK,CK}} \otimes \text{id}_{\text{CK}} \otimes \text{sp}_{\text{CK}} \otimes \text{sp}_{\text{CK}}\). We find a trace \(v \in \text{sp}_{\text{CK}} \otimes \text{id}_{\text{CK}} \otimes \text{sp}_{\text{CK}}\) such that \(u \mid Ck_4 + Ck_5 + Ck_6 + Ck_1 = v \mid Ck_4 + Ck_5 + Ck_6 + Ck_1\). We define \(v\) as follows.

- \(E_v = E_{u|Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}\)
- \(\lambda_v = \lambda_{u|Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}\)
- \(\preceq_v \Rightarrow \preceq_{u|Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1}\)
- \(\bigcirc_v\) is defined as follows, for all \(e, e' \in E_v\).

**JV1** If \(e, e' \in E_{u|Ck_2 + Ck_3 + Ck_1}\), then \(e \bigcirc_v e' \iff e \bigcirc_{u|e} e'\).

**JV2** If \(e \in E_{u|Ck_2}\) and \(e' \in E_{u|Ck_4}\), then \((\exists e'' \in E_{u|Ck_4}) (e \bigcirc_u e''\ and\ e'' \bigcirc_u e') \iff e \bigcirc_v e'\).

**JV3** If \(e \in E_{u|Ck_2}\) and \(e' \in E_{u|Ck_5}\), then \((\exists e_5 \in E_{u|Ck_5}) \exists e_3 \in E_{u|Ck_5} \exists e_1 \in E_{u|Ck_1}) (e_1 \bigcirc_u e\ and\ e_1 \bigcirc u e_3\ and\ e_3 \bigcirc u e_5\ or\ e_5 \bigcirc u e') \iff e \bigcirc_v e'\).

**JV4** If \(e \in E_{u|Ck_3}\) and \(e' \in E_{u|Ck_4}\), then \((\exists e'' \in E_{u|Ck_4}) (e \bigcirc_u e''\ and\ e'' \bigcirc_u e') \iff e \bigcirc_v e'\).

**JV5** The only pointers in \(\bigcirc_v\) are those specified by (JV1)–(JV4); that is, if \([e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_2} or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_5}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_6}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_8}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_9}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_10}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_11}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_12}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_13}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_14}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_15}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_16}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_17}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_18}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_19}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_20}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_21}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_22}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_23}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_24}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_25}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_26}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_27}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_28}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_29}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_30}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_31}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_32}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_33}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_34}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_35}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_36}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_37}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_38}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_39}]) or\ [e \in E_{u|Ck_1} and\ e' \in E_{u|Ck_40}]), then\ e \not\bigcirc_v e'.\)
We show that $v$ is justified, i.e. that $\varnothing_v : E_v \setminus \{e \in E_v \mid \lambda_v(e) \in I_{Ck_1}\} \rightarrow E_v$ is total and $(\forall e,e' \in E_v)(e \varnothing_v e' \Rightarrow (e \preceq_v e$ and $\lambda_v(e) \vdash \lambda_v(e'))).$ By inspection of the definition, we can see that the partial function $\varnothing_v$ is total because every non-initial event has a pointer. In particular, for any $e' \in E_{u\mid Ck_5}$, the existence of events $e,e_5,e_3,e_1$ in the definition is guaranteed since $u \upharpoonright Ck_2 + Ck_1 \in \text{id}_{Ck}$ and $u \upharpoonright Ck_3 + Ck_1 \in \text{id}_{Ck}$ and $u \upharpoonright Ck_5 + Ck'_3 \in \text{id}_{Ck}$.

Now, we show $(\forall e,e' \in E_v)(e \varnothing_v e' \Rightarrow (e \preceq_v e$ and $\lambda_v(e) \vdash \lambda_v(e'))).$ Suppose $e \varnothing_v e'$. By definition of $\varnothing_v$, we have the following cases.

- If $e,e' \in E_{u\mid Ck_2+Ck_3+Ck_1}$, then $e \varnothing_v e' \Rightarrow (e \preceq_v e$ and $\lambda_v(e) \vdash \lambda_v(e'))$.
- If $e \in E_{u\mid Ck_2}$ and $e' \in E_{u\mid Ck_4}$, then $e \varnothing_v e' \Rightarrow (\exists e'' \in E_{u\mid Ck_4})(e \varnothing_u e'' \land e'' \preceq_v e') \Rightarrow (\exists e'' \in E_{u\mid Ck_4})(e \preceq_u e''$ and $e'' \preceq_v e')$.
- If $e \in E_{u\mid Ck_5}$ and $e' \in E_{u\mid Ck_3}$, then $e \varnothing_v e' \Rightarrow (\exists e_5 \in E_{u\mid Ck_3}, \exists e_3 \in E_{u\mid Ck_3}, \exists e_1 \in E_{u\mid Ck_1})(e_1 \varnothing_u e$ and $e_1 \preceq_v e_3$ and $e_3 \varnothing_u e$ and $e_5 \varnothing_u e')$.
- If $e \in E_{u\mid Ck_4}$ and $e' \in E_{u\mid Ck_4}$, then $e \varnothing_v e' \Rightarrow (\exists e'' \in E_{u\mid Ck_4})(e \varnothing_u e'' \land e'' \preceq_v e') \Rightarrow (\exists e'' \in E_{u\mid Ck_4})(e \preceq_u e''$ and $e'' \preceq_v e')$.

Note that in each case $\lambda_v(e) \vdash \lambda_v(e')$.

If follows from its definition that $v$ satisfies $v \upharpoonright Ck_2 + Ck_1 \in \text{id}_{Ck}$ and $v \upharpoonright Ck_3 + Ck_1 \in \text{id}_{Ck}$. The fact that $v \upharpoonright Ck_4 + Ck_2 \in \text{id}_{Ck}$ follows from $u \upharpoonright Ck_4 + Ck_2 \in \text{id}_{Ck}$ and $u \upharpoonright Ck_4 + Ck'_4 \in \text{id}_{Ck}$ and the definition of $\varnothing_v$. The fact that $v \upharpoonright Ck_6 + Ck_3 \in \text{id}_{Ck}$ follows from $u \upharpoonright Ck'_6 + Ck_3 \in \text{id}_{Ck}$ and $u \upharpoonright Ck_6 + Ck'_3 \in \text{id}_{Ck}$ and the definition of $\varnothing_v$.

We still have to show $v \upharpoonright Ck_5 + Ck_2 \in \text{id}_{Ck}$. Let $e_5 \in E_{u\mid Ck_5}$. We have the following facts.

- Since $u \upharpoonright Ck_5 + Ck'_5 \in \text{id}_{Ck}$, there is $e'_5 \in E_{u\mid Ck'_5}$ such that $e_5 \equiv_{u\mid Ck_5+Ck'_5} e'_5$.
- As $u \upharpoonright Ck'_5 + Ck_3 \in \text{id}_{Ck}$, there is $e_3 \in E_{u\mid Ck_3}$ such that $e'_5 \equiv_{u\mid Ck'_5+Ck_3} e_3$.
- Because $u \upharpoonright Ck_3 + Ck_1 \in \text{id}_{Ck}$, there is $e_1 \in E_{u\mid Ck_1}$ such that $e_3 \equiv_{u\mid Ck_3+Ck_1} e_1$. 


6.1. CLOCK MONAD AND CLOCK MONOID

- Since \( u \upharpoonright Ck_2 + Ck_1 \in id_{Ck} \), there is \( e_2 \in E_{u|Ck_2} \) such that \( e_1 \equiv_{u|Ck_2 + Ck_1} e_2 \). Since \( \preceq_u \) is transitive, \( e_1 \approx_u e_2 \approx_u e_3 \approx_u e_5 \). So, for all \( e_5 \in E_{u|Ck_3} \) there is \( e_2 \in E_{u|Ck_2} \) such that \( e_5 \equiv_{u|Ck_3 + Ck_2} e_2 \). We use the same reasoning to conclude that for all \( e_2 \in E_{u|Ck_2} \) there is \( e_5 \in E_{u|Ck_3} \) such that \( e_5 \equiv_{u|Ck_3 + Ck_2} e_2 \). In essence, for all \( e \in E_{u|Ck_3 + Ck_2} \) there is \( e' \in E_{o|Ck_2 + Ck_3} \) such that \( e \equiv_{o|Ck_2 + Ck_3} e' \). Since \( E_v = E_{u|Ck_4 + Ck_5 + Ck_6 + Ck_2 + Ck_3 + Ck_1} \) and \( \lambda_v = \lambda_{u|Ck_4 + Ck_3 + Ck_2 + Ck_3 + Ck_1} \) and \( \preceq_v = \preceq_{u|Ck_4 + Ck_3 + Ck_2 + Ck_3 + Ck_1} \), we have for all \( e \in E_{v|Ck_3 + Ck_2} \), there is \( e' \in E_{v|Ck_2 + Ck_3} \) such that \( e \equiv_{v|Ck_3 + Ck_2} e' \).

We need to verify that \( \bowtie_{v|Ck_3 + Ck_2} \) conforms to Definition\[5.11\]. This can be noted by inspecting the definition of \( \bowtie_v \).

We now show that \( u \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_1 = v \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_1 \). From the definition of \( v \) and the definition of projection, \( E_{v|Ck_4 + Ck_5 + Ck_6 + Ck_1} = E_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} \) and \( \lambda_v = \lambda_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} \) and \( \preceq_v = \preceq_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} \). Next, we prove that for all \( e, e' \in E_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} \), we have \( e \bowtie_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \ implies \( e \bowtie_{v|Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \) through a case study on the labels of \( e, e' \).

- Case: \( \lambda_u(e) \in L_{Ck_1} \) and \( \lambda_u(e') \in L_{Ck_1} \). This is similar to the following case. We use (JV1) and (JV2).

- Case: \( \lambda_u(e) \in L_{Ck_1} \) and \( \lambda_u(e') \in L_{Ck_2} \). Let \( e \bowtie_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \). So, there is \( e_2 \in E_{u|Ck_2} \) such that \( e \bowtie_{u|Ck_2} e_2 \) and \( e_2 \bowtie_{u|Ck_1} e' \). Using (JV3) on \( e_2 \bowtie_{u|Ck_2} e' \), there is \( e_1 \in E_{u|Ck_1} \) such that \( e_1 \bowtie_{u|Ck_2} e_2 \) and \( e_1 \bowtie_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \). Using (JV1) on \( e \bowtie_{u|Ck_2} e_2 \), we get \( e \bowtie_{u|Ck_1} e' \). Therefore, \( e_1 = e \), so \( e \bowtie_{u|Ck_1} e_2 \bowtie_{u|Ck_1} e' \). For the opposite direction, let \( e \bowtie_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \). So, there are \( e_3 \in E_{u|Ck_3} \) such that \( e \bowtie_{u|Ck_3} e_3 \) and \( e_3 \bowtie_{u|Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \). Since \( u \upharpoonright Ck_2 + Ck_1 \in id_{Ck} \), there is \( e_2 \in E_{u|Ck_2} \) such that \( e \bowtie_{u|Ck_2} e_2 \). By (JV1), \( e \bowtie_{v|Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \). So, \( e \bowtie_{v|Ck_2} e' \). So, \( e \bowtie_{v|Ck_4 + Ck_5 + Ck_6 + Ck_1} e' \).

- Case: \( \lambda_u(e) \in L_{Ck_1} \) and \( \lambda_u(e') \in L_{Ck_2} \). This is similar to the previous case. We use (JV1) and (JV4).

- All other label assignments are illegal.

Next, we show that \( s_{Ck_2 + Ck_3, Ck_1} : id_{Ck} \otimes s_{Ck_2 + Ck_1} \supseteq s_{Ck} \otimes id_{Ck} \otimes s_{Ck} \).
Let $v \in sp_{Ck} \otimes id_{Ck} \otimes sp_{Ck}$. We find a trace $u \in a_{Ck,Ck,Ck} \otimes id_{Ck} \otimes sp_{Ck} \otimes sp_{Ck}$ such that $v \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_1 = u \upharpoonright Ck_4 + Ck_5 + Ck_6 + Ck_1$. We define $u$ using the same algorithm described in the proof of Lemma 3.24.

- $E_u = E_v|Ck_1 + Ck_2 + Ck_3 + Ck_4 + Ck_5 + Ck_6 + E_v|Ck_4 + Ck_5 + Ck_6$
- $\lambda_u = \lambda_v|Ck_1 + Ck_2 + Ck_3 + Ck_4 + Ck_5 + Ck_6 + \lambda_v|Ck_4 + Ck_5 + Ck_6$
- $\preceq_u$ is defined as follows for all $e, e' \in E_v$.

\begin{itemize}
  \item **P1** If $e, e' \in E_v|Ck_1 + Ck_2 + Ck_3$, then $e \preceq_v e' \iff in_1(e) \preceq_u in_1(e')$
  \item **P2** If $e \in E_v|Ck_1 + Ck_2 + Ck_3$ and $e' \in E_v|Ck_4 + Ck_5 + Ck_6$, then $e \preceq_v e' \iff in_1(e) \preceq_u in_1(e') \iff in_1(e) \preceq_u in_2(e')$
  \item **P3** If $e \in E_v|Ck_4 + Ck_5 + Ck_6$ and $e' \in E_v|Ck_1 + Ck_2 + Ck_3$, then $e \preceq_v e' \iff in_2(e) \preceq_u in_1(e') \iff in_1(e) \preceq_u in_1(e')$
  \item **P4** If $e, e' \in E_v|Ck_4 + Ck_5 + Ck_6$, then $e \preceq_v e' \iff in_1(e) \preceq_u in_1(e') \iff in_2(e) \preceq_u in_2(e') \iff in_1(e) \preceq_u in_1(e')$
\end{itemize}
6.1. CLOCK MONAD AND CLOCK MONOID

- if $[e \in E_0|Ck_4$ and $e' \in E_0|Ck_1$] or $[e \in E_0|Ck_4$ and $e' \in E_0|Ck_2$] or $[e \in E_0|Ck_4$ and $e' \in E_0|Ck_3$] or $[e \in E_0|Ck_5$ and $e' \in E_0|Ck_1$] or $[e \in E_0|Ck_5$ and $e' \in E_0|Ck_2$] or $[e \in E_0|Ck_5$ and $e' \in E_0|Ck_3$] or $[e \in E_0|Ck_6$ and $e' \in E_0|Ck_1$] or $[e \in E_0|Ck_6$ and $e' \in E_0|Ck_2$] or $[e \in E_0|Ck_6$ and $e' \in E_0|Ck_3$], then $in_1(e)$ and $in_2(e)$.

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 $\triangleleft_u$ is total because every non-initial event has a pointer. In particular, for any $e' \in E_u|Ck_5$, the existence of events $e, e_1, e_2$ in the definition is guaranteed since $v \upharpoonright Ck_2 + Ck_1 \in id_{Ck}$ and $u \upharpoonright Ck_3 + Ck_1 \in id_{Ck}$ and $u \upharpoonright Ck_5 + Ck_2 \in id_{Ck}$.

Now, we show $(\forall e, e' \in E_u)(e \triangleleft_u e' \Rightarrow (e \preceq_u e$ and $\lambda_u(e) \vdash \lambda_u(e'))).$ Suppose $e \triangleleft_u e'$. By definition of $\triangleleft_u$, we have the following cases.

- If $e, e' \in E_u|Ck_1+Ck_2+Ck_3$, then $e \triangleleft_u e' \Rightarrow in_1^{-1}(e) \triangleleft s in_1^{-1}(e')$ by (J1) \Rightarrow in_1^{-1}(e) \leq u e'$ by (P1)

- If $e \in E_u|Ck_4$ and $e' \in E_u|Ck_4$, then $e \triangleleft_u e' \Rightarrow in_2^{-1}(e) = in_1^{-1}(e')$ by (J2) \Rightarrow e \leq u e' by (P4)

- If $e \in E_u|Ck_5$ and $e' \in E_u|Ck_5$, then $e \triangleleft_u e' \Rightarrow in_2^{-1}(e) = in_1^{-1}(e')$ by (J3) \Rightarrow e \leq u e' by (P4)

- If $e \in E_u|Ck_6$ and $e' \in E_u|Ck_6$, then $e \triangleleft_u e' \Rightarrow in_2^{-1}(e) = in_1^{-1}(e')$ by (J4) \Rightarrow e \leq u e' by (P4)

- If $e \in E_u|Ck_2$ and $e' \in E_u|Ck_4$, then $e \triangleleft_u e' \Rightarrow in_1^{-1}(e) \triangleleft v in_2^{-1}(e')$ by (J2) \Rightarrow $in_1^{-1}(e) \leq v in_2^{-1}(e') \Rightarrow e \leq u e'$ by (P2)

- If $e \in E_u|Ck_3$ and $e' \in E_u|Ck_4$, then $e \triangleleft_u e' \Rightarrow (\exists e_1 \in E_u|Ck_1, \exists e_2 \in E_u|Ck_2)(e_1 \triangleleft v in_1^{-1}(e) \triangleleft v e_1 \triangleleft_v e_2$ and $e_2 \triangleleft v in_2^{-1}(e'))$ by (J3). Since $Ck$ is a single-event signature,
in each trace $s \in id_{Ck}$, we have by the definition of $id$ that $e \preceq_s e'$ implies $e \simeq_s e'$. As $v \upharpoonright Ck_2 + Ck_1 \in id_{Ck}$ and $v \upharpoonright Ck_3 + Ck_1 \in id_{Ck}$ and $u \upharpoonright Ck_2 + Ck_5 \in id_{Ck}$, we get $\text{in}_1^{-1}(e) \simeq_v e_1 \simeq_v e_2 \simeq_v \text{in}_2^{-1}(e') \therefore e \preceq_u e'$ by (P2)

- If $e \in E_{u|Ck_3}$ and $e' \in E_{u'|Ck_4}$, then $e \preceq_u e' \therefore \text{in}_1^{-1}(e) \preceq_v \text{in}_2^{-1}(e')$ by (J4). \therefore \text{in}_1^{-1}(e) \preceq_v \text{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 $u \in \kappa_{Ck,Ck,Ck} \iota id_{Ck} \otimes sp_{Ck} \iota sp_{Ck}$.

- $u \upharpoonright Ck_2 + Ck_1 \in id_{Ck}$ and $u \upharpoonright Ck_3 + Ck_1 \in id_{Ck}$ follow from (P1), (J1) and the fact that $v \upharpoonright Ck_2 + Ck_1 \in id_{Ck}$ and $v \upharpoonright Ck_3 + Ck_1 \in id_{Ck}$.

- $u \upharpoonright Ck_4 + Ck_2 \in id_{Ck}$ follows from (P2), (P3), (J2) and $v \upharpoonright Ck_4 + Ck_2 \in id_{Ck}$.

- $u \upharpoonright Ck_6 + Ck_3 \in id_{Ck}$ follows from (P2), (P3), (J4) and $v \upharpoonright Ck_6 + Ck_3 \in id_{Ck}$.

- $u \upharpoonright Ck_4 + Ck_4 \in id_{Ck}$ follows from (P4), (J2) and the fact that $\preceq_v$ is reflexive.

- $u \upharpoonright Ck_5 + Ck_5 \in id_{Ck}$ follows from (P4), (J3) and the fact that $\preceq_v$ is reflexive.

- $u \upharpoonright Ck_6 + Ck_6 \in id_{Ck}$ follows from (P4), (J4) and the fact that $\preceq_v$ is reflexive.

We still have to show $u \upharpoonright Ck_5 + Ck_3 \in id_{Ck}$. Let $e \in E_{u|Ck_5}$. It follows that there is $e_5 \in E_{u|Ck_5}$ such that $e = \text{in}_2(e_5)$ and $\text{in}_1(e_5) \in u \upharpoonright Ck_5$ and $e \simeq_u \text{in}_1(e_5)$ and $e \preceq_u Ck_5 + Ck_5 \text{ in}_1(e_5)$. Let us refer to the previous fact by (\textcircled{4}). For all $e_5 \in E_{v|Ck_5}$, we have the following facts.

- Since $v \upharpoonright Ck_5 + Ck_2 \in id_{Ck}$, there is $e_2 \in E_{v|Ck_2}$ such that $e_5 \equiv_{v|Ck_5+Ck_5} e_2$.

- As $v \upharpoonright Ck_2 + Ck_1 \in id_{Ck}$, there is $e_1 \in E_{v|Ck_1}$ such that $e_2 \equiv_{u|Ck_2+Ck_1} e_1$.

- Because $v \upharpoonright Ck_3 + Ck_1 \in id_{Ck}$, there is $e_3 \in E_{v|Ck_1}$ such that $e_1 \equiv_{v|Ck_3+Ck_1} e_3$.

Since $\preceq_v$ is transitive, $e_2 \simeq_v e_5$. We use (P2) and (P3) to find that $e_2 \simeq_u \text{in}_1(e_5)$ and $e_2 \simeq_u \text{in}_2(e_5)$.

We use (\textcircled{4}) to find that for all $e \in E_{u|Ck_5}$ there is $e_2 \in E_{u|Ck_2}$ such that $e \equiv_{u|Ck_5+Ck_2} e_2$. We use the same reasoning to conclude that for all $e_2 \in E_{u|Ck_2}$ there is $e \in E_{u|Ck_5}$ such that $e \equiv_{u|Ck_5+Ck_2} e_2$. In essence, for all $e \in E_{u|Ck_5+Ck_2}$, there is $e' \in E_{u|Ck_5+Ck_2}$ such that $e \equiv_{u|Ck_5+Ck_2} e'$. 

We now show that \( \bigwedge_{\mathcal{O}} \mathcal{C}_{k_3 + k_3} \) conforms to Definition 5.11. This can be noted by inspecting the definition of \( \bigwedge_u \).

We now need to show that \( u \uparrow \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 = v \uparrow \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \). From the definition of \( u \) and the definition of projection, we see that \( E_u \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 = in_1(E_v \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1) \) and \( \lambda_u \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 = \lambda_v \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \circ in_1 \). We can also verify that \((\forall e, e' \in E_v \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1) (e \preceq_v \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \Leftrightarrow in_1(e) \preceq_u \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \circ in_1(e')) \). Next, we prove that for all \( e, e' \in E_v \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \), we have \( e \bigwedge_{\mathcal{O}} \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \; e' \Leftrightarrow in_1(e) \bigwedge_{\mathcal{O}} \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \; in_1(e') \) through a case study on the labels of \( e, e' \):

- **Case:** \( \lambda_u(e) \in L_{\mathcal{C}_k} \) and \( \lambda_u(e') \in L_{\mathcal{C}_k} \). This is similar to the following case. We use (J1) and (J2).

- **Case:** \( \lambda_u(e) \in L_{\mathcal{C}_k} \) and \( \lambda_u(e') \in L_{\mathcal{C}_k} \). Let \( e \bigwedge_{\mathcal{O}} \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \; e' \). So, there are \( e_3 \in E_u \mathcal{C}_3 \) and \( e_5 \in E_u \mathcal{C}_5 \) such that \( e \bigwedge_u e_3 \) and \( e_3 \bigwedge_u e_5 \) and \( e_5 \bigwedge_u e' \). Using (J3) on \( e_3 \bigwedge_u e_5 \), there are \( e_1 \in E_v \mathcal{C}_1 \), \( e_2 \in E_v \mathcal{C}_2 \) such that \( e_1 \bigwedge_v in_1^{-1}(e_3) \) and \( e_1 \bigwedge_v e_2 \) and \( e_2 \bigwedge_v in_2^{-1}(e_5) \). Using (J1) on \( e \bigwedge_u e_3 \), we get \( in_1^{-1}(e) \bigwedge_v in_1^{-1}(e_3) \). Therefore, \( e_1 = in_1^{-1}(e) \). Using (J3) on \( e_5 \bigwedge_u e' \), we get \( in_2^{-1}(e) = in_1^{-1}(e') \). So, \( in_1^{-1}(e) \bigwedge_v e_2 \) and \( e_2 \bigwedge_v in_1^{-1}(e') \). So, \( in_1^{-1}(e) \bigwedge_v in_1^{-1}(e') \). For the opposite direction, let \( e \bigwedge_v e_2 \) and \( e_2 \bigwedge_v e' \). Since \( v \uparrow \mathcal{C}_3 + \mathcal{C}_1 \) is id\( \mathcal{C}_k \), there is \( e_3 \in E_v \mathcal{C}_3 \) such that \( e \bigwedge_v e_3 \). By (J1), \( in_1(e) \bigwedge_u in_1(e_3) \). By (J3), \( in_1(e_3) \bigwedge_u in_2(e') \) and \( in_2(e') \bigwedge_u in_1(e') \). So, \( in_1(e) \bigwedge_v \mathcal{C}_4 + \mathcal{C}_5 + \mathcal{C}_6 + \mathcal{C}_1 \; in_1(e') \).

- **Case:** \( \lambda_u(e) \in L_{\mathcal{C}_k} \) and \( \lambda_u(e') \in L_{\mathcal{C}_k} \). 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 $u \in \alpha_{Ck, Ck, Ck} \otimes id_{Ck} \otimes sp_{Ck}; sp_{Ck}$ 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.

Proof. We need to prove that \( \gamma_{Ck_1,Ck_2} \cdot sp_{Ck} = sp_{Ck} \).

We first show that \( \forall\ E \in \Theta \cup \Theta \cup \Theta \), \( \forall\ e \in E \), \( \exists\ \gamma(p) \in \gamma(p) \) such that \( \gamma(p) \cup \gamma(p) \) is defined as follows, for all \( e, e' \in E \).

- \( E_v = E_{v[Ck_2 + Ck_3 + Ck_1]} \)
- \( \lambda_v = \lambda_{u[Ck_2 + Ck_3 + Ck_1]} \)
- \( \preceq_v = \preceq_{u[Ck_2 + Ck_3 + Ck_1]} \)
- \( \cdot_v \) is defined as follows, for all \( e, e' \in E_u \).

1. If \( e \in E_u[Ck_1] \) and \( e' \in E_u[Ck_2] \), then \( (\exists e'' \in E_{u[Ck_2]})(e \cdot_v e'' \leftrightarrow e' \cdot_v e') \).
2. If \( e \in E_u[Ck_3] \) and \( e' \in E_u[Ck_2] \), then \( (\exists e'' \in E_{u[Ck_2]})(e \cdot_v e'' \leftrightarrow e' \cdot_v e') \).
3. The only pointers in \( \cdot_v \) are those specified by (JV1)–(JV2); that is, if \( e \in E_{u[Ck_2]} \) and \( e' \in E_{u[Ck_1]} \) or \( e \in E_{u[Ck_3]} \) and \( e' \in E_{u[Ck_1]} \).

We first show that \( v \) is justified, i.e. that \( \cdot_v \colon E_v \setminus \{e \in E_v \mid \lambda_v(e) \in I_{Ck_1}\} \rightarrow E_v \) is total and \( (\forall e, e' \in E_v)(e \cdot_v e' \leftrightarrow (e \preceq_v e \land \lambda_v(e) \vdash \lambda_v(e')) \). By inspection of the definition, we can see that the partial function \( \cdot_v \) is total because every non-initial event has a pointer. In particular, for any \( e' \in E_u[Ck_2] \), the existence of events \( e'' \) in the definition is guaranteed since \( u \cup Ck_2 + Ck_1 \in id_{Ck} \) and \( u \cup Ck_2 + Ck_2' \in id_{Ck} \). The same is true for events \( e' \in E_u[Ck_2] \).

Now, we show \( (\forall e, e' \in E_v)(e \cdot_v e' \leftrightarrow (e \preceq_v e \land \lambda_v(e) \vdash \lambda_v(e')) \). Suppose \( e \cdot_v e' \). By definition of \( \cdot_v \), we have the following cases.

- If \( e \in E_u[Ck_1] \) and \( e' \in E_u[Ck_2] \), then \( e \cdot_v e' \).
- If \( e \in E_u[Ck_2] \) and \( e' \in E_u[Ck_3] \), then \( e \cdot_v e' \).
- If \( e \in E_u[Ck_3] \) and \( e' \in E_u[Ck_2] \), then \( e \cdot_v e' \).
- If \( e \in E_u[Ck_1] \) and \( e' \in E_u[Ck_1] \), then \( e \cdot_v e' \).
- If \( e \in E_u[Ck_2] \) and \( e' \in E_u[Ck_2] \), then \( e \cdot_v e' \).
- If \( e \in E_u[Ck_3] \) and \( e' \in E_u[Ck_3] \), then \( e \cdot_v e' \).
CHAPTER 6. GLOBAL CLOCKS AND DETERMINISM

- If \( e \in E_u|Ck_1 \) and \( e' \in E_u|Ck_3 \), then \( e \bowtie_v e' \) : \( (\exists e'' \in E_u|Ck_3)(e \bowtie_u e'' \text{ and } e'' \bowtie_u e') \) .

Next, we prove that for all \( e, e' \), there exists \( \lambda \) such that \( \lambda \bowtie v e \bowtie v e' \).

We now need to show that \( \lambda \bowtie v e \bowtie v e' \) 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 \( e \bowtie_u Ck_2 \), such that \( e \bowtie_u e_2 \) and \( e_2 \bowtie_u e' \). Using (J1) we get \( e \bowtie_v e' \). For the opposite direction, let \( e \bowtie_v e' \). So, there is \( e_2 \in E_u|Ck_2 \) such that \( e \bowtie_u e_2 \) and \( e_2 \bowtie_u e' \). By definition of projection, \( e \bowtie_u Ck_2 \).

- 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 (J2).

- All other label assignments are illegal.

Next, we show that \( \gamma_{Ck_1,Ck_3}; sp_{Ck} \supseteq sp_{Ck} \). Let \( v \in sp_{Ck} \). We find a trace \( u \in \gamma_{Ck_1,Ck_3}; sP_{Ck} \) such that \( v = u \bowtie Ck_2 + Ck_3 + Ck_1 \). We define \( u \) using the same algorithm described in the second half of the proof of equation (6.1) in Lemma 6.4

- \( E_u = E_v + E_v|Ck_2+Ck_3 \)

- \( \lambda u = \lambda v + \lambda v|Ck_2+Ck_3 \)

- \( \leq u \) is defined as follows for all \( e, e' \in E_v \).

P1 If \( e, e' \in E_v \), then \( e \leq_v e' \iff in_1(e) \leq u in_1(e') \)

P2 If \( e \in E_v \) and \( e' \in E_v|Ck_2+Ck_3 \), then \( e \leq_v e' \iff in_1(e) \leq u in_1(e') \iff in_1(e) \leq u in_2(e') \)

P3 If \( e \in E_v|Ck_2+Ck_3 \) and \( e' \in E_v \), then \( e \leq_v e' \iff in_2(e) \leq u in_1(e') \iff in_1(e) \leq u in_1(e') \)

P4 If \( e, e' \in E_v|Ck_2+Ck_3 \), then \( e \leq_v e' \iff in_1(e) \leq u in_1(e') \iff in_2(e) \leq u in_2(e') \iff in_1(e) \leq u in_2(e') \iff in_2(e) \leq u in_1(e') \)
6.1. CLOCK MONAD AND CLOCK MONOID

- \( \trianglelefteq_u \) is defined as follows, for all \( e, e' \in E_o \).

**J1** If \( e \in E_o|\text{Ck}_k \) and \( e' \in E_o|\text{Ck}_k \), then \( e \trianglelefteq_v e' \iff \text{in}_1(e) \trianglelefteq u \text{in}_2(e') \iff \text{in}_2(e') \trianglelefteq u \text{in}_1(e') \).

**J2** If \( e \in E_o|\text{Ck}_k \) and \( e' \in E_o|\text{Ck}_k \), then \( e \trianglelefteq_v e' \iff \text{in}_1(e) \trianglelefteq u \text{in}_2(e') \iff \text{in}_2(e') \trianglelefteq u \text{in}_1(e') \).

**J3** The only pointers in \( \trianglelefteq_v \) are those specified by (J1)–(J2); that is,

- if \( e \in E_o|\text{Ck}_k \) and \( e' \in E_o|\text{Ck}_k \) or \( e \in E_o|\text{Ck}_k \) and \( e' \in E_o|\text{Ck}_k \), then \( \text{in}_1(e) \not\trianglelefteq_v \text{in}_1(e') \) and \( \text{in}_2(e) \not\trianglelefteq_v \text{in}_1(e') \).
- if \( e, e' \in E_o|\text{Ck}_k \) or \( e, e' \in E_o|\text{Ck}_k \), then \( \text{in}_1(e) \not\trianglelefteq_v \text{in}_1(e') \) and \( \text{in}_2(e) \not\trianglelefteq_v \text{in}_2(e') \).
- if \( e \in E_o|\text{Ck}_k \) and \( e' \in E_o|\text{Ck}_k \) or \( e \in E_o|\text{Ck}_k \) and \( e' \in E_o|\text{Ck}_k \), then \( \text{in}_1(e) \not\trianglelefteq_v \text{in}_2(e') \) and \( \text{in}_2(e) \not\trianglelefteq_v \text{in}_2(e') \).

We can show that \( \trianglelefteq_u \) 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 \( u \in \gamma_{\text{Ck}_k, \text{Ck}_k} \delta \supset Ck \).

- \( u \mid \text{Ck}_k + \text{Ck}_k' \in \text{id}_Ck \) follows from (P4), (J1) and the fact that \( \trianglelefteq_v \) is reflexive.
- \( u \mid \text{Ck}_k + \text{Ck}_k' \in \text{id}_Ck \) follows from (P4), (J2) and the fact that \( \trianglelefteq_v \) is reflexive.
- \( u \mid \text{Ck}_k' + \text{Ck}_1 \in \text{id}_Ck \) follows from (P1), (P4), (J1) and the fact that \( v \mid \text{Ck}_k + \text{Ck}_1 \in \text{id}_Ck \).
- \( u \mid \text{Ck}_k' + \text{Ck}_1 \in \text{id}_Ck \) follows from (P1), (P4), (J2) and the fact that \( v \mid \text{Ck}_k + \text{Ck}_1 \in \text{id}_Ck \).

We now show that \( \trianglelefteq_{u|\text{Ck}_k + \text{Ck}_k} \) conforms to Definition [5.11](#). This can be noted by inspecting the definition of \( \trianglelefteq_u \).

Next, we need to show that \( u \mid \text{Ck}_k + \text{Ck}_k + \text{Ck}_1 = v \). From the definition of \( u \) and the definition of projection, we see that \( E_{u|\text{Ck}_k + \text{Ck}_k + \text{Ck}_1} = \text{in}_1(E_o) \) and \( \lambda_{u|\text{Ck}_k + \text{Ck}_k + \text{Ck}_1} = \lambda_{v \circ \text{in}_1} \). We can also verify that \((\forall e, e' \in E_o|\text{Ck}_k + \text{Ck}_k + \text{Ck}_1)(e \trianglelefteq_v e' \iff \text{in}_1(e) \trianglelefteq u \text{in}_1(e'))\). Next, we prove that for all \( e, e' \in E_o|\text{Ck}_k + \text{Ck}_k + \text{Ck}_1 \), we have \( e \trianglelefteq_v e' \iff \text{in}_1(e) \trianglelefteq u \text{in}_1(e') \) through a case study on the labels of \( e, e' \).
CHAPTER 6. GLOBAL CLOCKS AND DETERMINISM

• Case: \( \lambda_v(e) \in L_{Ck_1} \) and \( \lambda_v(e') \in L_{Ck_2} \). Let \( e \prec_v e' \). By (J1), \( in_1(e) \prec_u in_2(e') \) and \( in_2(e') \prec_u in_1(e') \). By definition of projection, \( e \prec u_{|Ck_2+Ck_3+Ck_1} e' \). For the opposite direction, let \( e \prec u_{|Ck_2+Ck_3+Ck_1} e' \). So, there is \( e_2 \in E_{u_{|Ck_2}} \) such that \( e \prec u e_2 \) and \( e_2 \prec u e' \). Using (J1) we get \( e \prec_v e' \).

• 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 \( \alpha \).

\textbf{Lemma 6.6.} Let \( \sigma : A \to B, \tau : C \to D, v : E \to F \) be causal processes. We have \( \sigma \otimes (\tau \otimes v) = \alpha_{A,C,E}^{-1} \sigma \otimes \tau \otimes v \); \( \alpha_{B,D,F} \) and \( (\sigma \otimes \tau) \otimes v = \alpha_{A,C,E} \sigma \otimes (\tau \otimes v) ; \alpha_{B,D,F}^{-1} \).

The other two appear in the literature.

\textbf{Lemma 6.7 ([JS93, Prop. 1.1])}. \( \rho_{A \otimes B} = \alpha_{A,B,I} ; id_A \otimes \rho_B \).

\textbf{Lemma 6.8 ([JS93, Prop. 1.1])}. \( \lambda_{A \otimes B} = \alpha_{I,A,B}^{-1} \lambda_A \otimes id_B \).

\textbf{Lemma 6.9}. \( \langle T, \eta, \mu \rangle \) is a monad.

\textit{Proof.} We need to prove the following.

\begin{align*}
T(\mu_A) ; \mu_A &= \mu_{TA} ; \mu_A \quad (6.5) \\
T(\eta_A) ; \mu_A &= id_{TA} \quad (6.6) \\
\eta_{TA} ; \mu_A &= id_{TA} \quad (6.7)
\end{align*}

For (6.5), we have

\begin{align*}
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,Ck} ; id_A \otimes (sp_{Ck} \otimes id_{Ck}) ; \alpha_{A,Ck,Ck} ; \alpha_{A,Ck,Ck} ; id_A \otimes sp_{Ck} \quad \text{by Lemma 6.6} \\
&= id_A \otimes sp_{Ck} \quad \text{by Lemma 6.6} \\
&= \alpha_{A,Ck,Ck} \otimes id_{Ck} ; \alpha_{A,Ck,Ck} ; id_A \otimes (sp_{Ck} \otimes id_{Ck}) ; id_A \otimes sp_{Ck}
\end{align*}
6.1. CLOCK MONAD AND CLOCK MONOID

\[ T(\eta_A); \mu_A = (\rho_A^{-1}; id_A \otimes sp_Ck) \otimes \alpha_{A,ck,ck}; id_A \otimes sp_Ck \] by definition

\[ = \rho_A^{-1} \otimes id_Ck; (id_A \otimes sp_Ck) \otimes \alpha_{A,ck,ck}; id_A \otimes sp_Ck \] by functoriality of \( \otimes \)

\[ = \rho_A^{-1} \otimes id_Ck; \alpha_{A,ck,ck}; id_A \otimes (cs_Ck \otimes id_Ck); \alpha_{A,ck,ck}^{-1}; \alpha_{A,ck,ck}; id_A \otimes sp_Ck \] by Lemma 6.6

\[ = \rho_A^{-1} \otimes id_Ck; \alpha_{A,ck,ck}; id_A \otimes (cs_Ck \otimes id_Ck); \alpha_{A,ck,ck}^{-1}; \alpha_{A,ck,ck}; id_A \otimes sp_Ck \] by Lemma 6.6

\[ = \rho_A^{-1} \otimes id_Ck; \alpha_{A,ck,ck}; id_A \otimes (cs_Ck \otimes id_Ck); sp_Ck \] by functoriality of \( \otimes \)

\[ = \rho_A^{-1} \otimes id_Ck; \alpha_{A,ck,ck}; id_A \otimes \lambda_Ck \] monoid laws

\[ = \rho_A^{-1} \otimes id_Ck; \rho_A \otimes id_Ck \] coherence of \( \rho \)

\[ = (\rho_A^{-1}; \rho_A) \otimes id_Ck \] by functoriality of \( \otimes \)

\[ = id_A \otimes id_Ck \]

\[ = id_A \otimes sp_Ck \] by functoriality of \( \otimes \)

\[ = id_{TA} \]

For (6.6), we have

\[ \eta_{TA}; \mu_A = \rho_A^{-1}; id_A \otimes sp_Ck \] by definition

\[ = \rho_A^{-1} \otimes id_Ck; (id_A \otimes sp_Ck) \otimes \alpha_{A,ck,ck}; id_A \otimes sp_Ck \] by functoriality of \( \otimes \)

\[ = \rho_A^{-1} \otimes id_Ck; \alpha_{A,ck,ck}; id_A \otimes (cs_Ck \otimes id_Ck); \alpha_{A,ck,ck}^{-1}; \alpha_{A,ck,ck}; id_A \otimes sp_Ck \] by Lemma 6.6

\[ = \rho_A^{-1} \otimes id_Ck; \alpha_{A,ck,ck}; id_A \otimes (cs_Ck \otimes id_Ck); sp_Ck \] by functoriality of \( \otimes \)

\[ = \rho_A^{-1} \otimes id_Ck; \alpha_{A,ck,ck}; id_A \otimes \lambda_Ck \] monoid laws

\[ = \rho_A^{-1} \otimes id_Ck; \rho_A \otimes id_Ck \] coherence of \( \rho \)

\[ = (\rho_A^{-1}; \rho_A) \otimes id_Ck \] by functoriality of \( \otimes \)

\[ = id_A \otimes id_Ck \]

\[ = id_A \otimes sp_Ck \] by functoriality of \( \otimes \)

\[ = id_{TA} \]
Theorem 6.12. \langle T, \eta, \mu, t \rangle is a commutative strong monad.

Proof. We need to prove the following.

\begin{align*}
&= \rho_{A \otimes Ck}^{-1}; \alpha_{A,Ck} \circ \id_A \otimes (\id_{Ck} \otimes cs_{Ck}) ; \alpha_{A,Ck,Ck}^{-1} ; \alpha_{A,Ck,Ck} ; \id_A \otimes s p_{Ck} \quad \text{by Lemma 6.6} \\
&= \rho_{A \otimes Ck}^{-1}; \alpha_{A,Ck} \circ \id_A \otimes (\id_{Ck} \otimes cs_{Ck}) ; \id_A \otimes s p_{Ck} \\
&= \rho_{A \otimes Ck}^{-1}; \alpha_{A,Ck} \circ \id_A \otimes (\id_{Ck} \otimes cs_{Ck}; s p_{Ck}) \quad \text{by functoriality of } \otimes \\
&= \rho_{A \otimes Ck}^{-1}; \alpha_{A,Ck} \circ \id_A \otimes \rho_{Ck} \quad \text{monoid laws} \\
&= \rho_{A \otimes Ck} ; \rho_{A \otimes Ck} \quad \text{by Lemma 6.7} \\
&= \id_{TA} \\
&= \id_{TA}
\end{align*}

We define the component of the strength of the clock monad as \( t_{A,B} = \alpha_{A,B,Ck}^{-1} \). 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 \( \gamma \) \[IS93\].

Lemma 6.10. \( t'_{A,B} = \gamma_{A \otimes Ck,B} ; \alpha_{A,B,Ck}^{-1} ; \gamma_{B,A} \otimes \id_{Ck} = \alpha_{A,Ck,B} ; \id_A \otimes \gamma_{Ck,B} ; \alpha_{A,B,Ck}^{-1} \).

Lemma 6.11. \( \gamma_{A,B \otimes C} = \alpha_{A,B,C}^{-1} ; \gamma_{A,B} \otimes \id_C ; \alpha_{B,A,C} ; \id_B \otimes \gamma_{A,C} ; \alpha_{B,C,A}^{-1} \).

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.

\begin{align*}
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} \quad (6.8) \\
t_{I,A} ; T(\lambda_A) &= \lambda_{TA} \quad (6.9) \\
\id_A \otimes \eta_B ; t_{A,B} &= \eta_{A \otimes B} \quad (6.10) \\
\id_A \otimes \mu_B ; t_{A,B} &= t_{A,TB} ; T(t_{A,B}) ; \mu_{A \otimes B} \quad (6.11) \\
t_{TA,B} ; T(t'_{A,B}) ; \mu_{A \otimes B} &= t'_{A,TB} ; T(t_{A,B}) ; \mu_{A \otimes B} \quad (6.12)
\end{align*}

For (6.8), we have

\begin{align*}
LHS &= \alpha_{A,B,C}^{-1} ; \alpha_{A,B,C} \otimes \id_{Ck} \\
&= \alpha_{A,B,C} ; \id_A \otimes \alpha_{B,C,k}^{-1} ; \alpha_{A,B,C,k}^{-1} \quad \text{by coherence of } \alpha \\
&= RHS
\end{align*}
For (6.11), we have
\[
LHS = \id_A \otimes (\alpha_{A,B,Ck}^{-1}; \id_B \otimes \cs_{Ck}); \alpha_{A,B,Ck}^{-1}
\]

\[
= \id_A \otimes \id_B \otimes \cs_{Ck}; \alpha_{A,B,Ck}^{-1} \quad \text{by functoriality of } \otimes
\]

\[
= \id_A \otimes \id_B \otimes \cs_{Ck}; \alpha_{A,B,Ck}^{-1} \quad \text{by Lemma 6.6}
\]

\[
= \id_A \otimes \id_B \otimes \cs_{Ck} \quad \text{by functoriality of } \otimes
\]

\[
= \rho_{A \otimes B}^{-1}; \id_{A \otimes B} \otimes \cs_{Ck} \quad \text{by Lemma 6.7}
\]

\[
= RHS
\]

For (6.12), we have
\[
RHS = \gamma_{A \otimes Ck, B \otimes Ck}; \alpha_{A,B,Ck}^{-1}; \gamma_{B \otimes Ck, A \otimes Ck}; \id_{Ck}; \alpha_{A,B,Ck}^{-1}; \id_{Ck}; \alpha_{A,B,\id_{Ck}}; \id_{A \otimes B} \otimes \cs_{Ck}
\]

\[
= \alpha_{A,B,Ck}; \id_{A \otimes B} \otimes \cs_{Ck} \quad \text{by Lemma 6.11}
\]

\[
= \alpha_{A,B,Ck}; \id_{A \otimes B} \otimes \cs_{Ck} \quad \text{by Lemma 6.11}
\]

\[
= \alpha_{A,B,Ck}; \id_{A \otimes B} \otimes \cs_{Ck} \quad \text{by functoriality of } \otimes
\]

\[
= \alpha_{A,B,Ck}; \id_{A \otimes B} \otimes \cs_{Ck} \quad \text{by Lemma 6.7}
\]

\[
= \alpha_{A,B,Ck}; \id_{A \otimes B} \otimes \cs_{Ck}
\]
$\alpha_{A \otimes B, C K, C K} \cdot id_{A \otimes B} \otimes sp_{C K}$  by Lemma 6.6

$= \alpha_{A \otimes C K, B, C K}^{-1} \cdot id_{A} \otimes \gamma_{C K, B} \cdot \alpha_{A, C K, B}^{-1} \cdot id_{C K} \cdot \alpha_{A \otimes B, C K, C K} \cdot id_{A \otimes B} \otimes \gamma_{C K, C K}$

$\cdot id_{(A \otimes B) \otimes (C K \otimes C K)} \cdot id_{A \otimes B} \otimes sp_{C K}$

by functoriality of $\otimes$ and the following equalities stemming from the coherence of $\alpha$,

1. $\alpha_{A, C K, B} \cdot id_{A} \otimes \alpha^{-1}_{A \otimes C K, B, C K} \cdot \alpha_{A, C K, B}^{-1} = \alpha_{A \otimes C K, B, C K}^{-1} \cdot \alpha_{A, C K, B}$ and

2. $\alpha_{A, B \otimes C K, C K} \cdot id_{A} \otimes \alpha_{B, C K, C K} \cdot \alpha^{-1}_{A, B \otimes C K, C K} = \alpha_{A, B, C K}^{-1} \cdot \alpha_{A \otimes B, C K, C K}$ and

3. $\alpha_{A, B, C K, C K} \cdot id_{A} \otimes \alpha^{-1}_{B, C K, C K} \cdot \alpha_{A, B, C K}^{-1} \cdot id_{C K} \cdot \alpha_{A \otimes B, C K, C K} = id_{(A \otimes B) \otimes (C K \otimes C K)}$

We have

$LHS = \alpha_{A \otimes C K, B, C K}^{-1} \cdot (\gamma_{A \otimes C K, B} \cdot \alpha_{A, C K, B}^{-1} \cdot \gamma_{B, A} \otimes id_{C K}) \cdot id_{C K} \cdot \alpha_{A \otimes B, C K, C K} \cdot id_{A \otimes B} \otimes sp_{C K}$

$= \alpha_{A \otimes C K, B, C K}^{-1} \cdot (\alpha_{A, C K, B} \cdot id_{A} \otimes \gamma_{C K, B} \cdot \alpha_{A, B, C K}^{-1}) \cdot id_{C K} \cdot \alpha_{A \otimes B, C K, C K} \cdot id_{A \otimes B} \otimes sp_{C K}$  by Lemma 6.10

$= \alpha_{A \otimes C K, B, C K}^{-1} \cdot (\alpha_{A, C K, B} \otimes id_{C K} \cdot id_{A} \otimes \gamma_{C K, B} \cdot \alpha_{A, B, C K}^{-1} \cdot id_{C K}) \cdot \alpha_{A \otimes B, C K, C K} \cdot id_{A \otimes B} \otimes sp_{C K}$

by functoriality of $\otimes$

$= \alpha_{A \otimes C K, B, C K}^{-1} \cdot (\alpha_{A, C K, B} \otimes id_{C K} \cdot id_{A} \otimes \gamma_{C K, B} \cdot \alpha_{A, B, C K}^{-1} \cdot id_{C K}) \cdot \alpha_{A \otimes B, C K, C K} \cdot id_{A \otimes B} \otimes \gamma_{C K, C K}$

$\cdot id_{A \otimes B} \otimes sp_{C K}$  by commutativity of the monoid

$= \alpha_{A \otimes C K, B, C K}^{-1} \cdot (\alpha_{A, C K, B} \otimes id_{C K} \cdot id_{A} \otimes \gamma_{C K, B} \cdot \alpha_{A, B, C K}^{-1} \cdot id_{C K}) \cdot \alpha_{A \otimes B, C K, C K} \cdot id_{A \otimes B} \otimes \gamma_{C K, C K}$

$\cdot id_{A \otimes B} \otimes sp_{C K}$  by functoriality of $\otimes$

$= RHS$

The commutative strong monad $(T, \eta, \mu, t)$ induces a symmetric monoidal monad $(T, \eta, \mu, m)$, 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.

$m_{A,B} = t_{TA,B} \cdot T(t'_{A,B}) \cdot \mu_{A \otimes B} \equiv id_{A} \otimes id_{B} \otimes sp_{C K}$

$TA \otimes TB \xrightarrow{t_{TA,B}} T(TA \otimes B) \xrightarrow{T(t'_{A,B})} T(T(A \otimes B)) \xrightarrow{\mu_{A \otimes B}} T(A \otimes B)$
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 $\text{SynProc}_T$ is defined as follows.

- The objects of $\text{SynProc}_T$ are those of $\text{SynProc}_P$.
- The set of morphisms $\text{hom}_{\text{SynProc}_T}(A, B)$ is $\text{hom}_{\text{SynProc}_P}(A, TB)$.
- 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 $\eta_A$.

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 $\mathcal{C}$, then the Kleisli category $\mathcal{C}_T$ 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.](image-url)
Corollary 6.14. \( \text{SynProc}_T \) is a symmetric monoidal category.

We will identify a subcategory \( \text{SynProc}' \) of \( \text{SynProc}_T \) wherein for each morphism \( \sigma : A \rightarrow B \), 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 \( \sigma : A \rightarrow B \) in \( \text{SynProc}_T \) is called pure when all \( s \in \sigma \) satisfy the following. For all \( e \in E_s \downarrow A \), if \( e' \prec e \) 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 \( \text{SynProc}_T \). 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 \( \text{SynProc}_T \) is pure by definition.

Lemma 6.16. If \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \) in \( \text{SynProc}_T \) 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 \( u \in \sigma ; T \tau \otimes id_{Ck} \circ \mu_C \) over \( \Theta_f((A \Rightarrow B \otimes Ck_4) \Rightarrow (C \otimes Ck_2) \otimes Ck_3) \Rightarrow C' \otimes Ck_1) \) such that \( u \upharpoonright A + C' + \)

Figure 6.8: A circuit representation of composition in the Kleisli category \( \text{SynProc}_T \).

Figure 6.9: A circuit representation of \( (id_B \otimes T \tau) ; T \text{eval}_{B,C} \).
6.2. CLOCKED PROCESSES

\( Ck_1 = v \). Suppose \( e' \triangleright v e \) and \( \lambda_v(e) \in I_A \). By definition of composition, there are events 
\( e_1 \in E_u|C+\text{Ck}_2+\text{Ck}_3, e_2 \in E_u|B+\text{Ck}_4 \) such that 
\( \lambda_u(e_1) \in I_C \otimes \text{Ck}_2 \otimes \text{Ck}_3 \) and \( \lambda_u(e_2) \in I_B \otimes \text{Ck}_4 \) and \( e' \triangleright u e_1 \) 
and \( e_1 \triangleright u e_2 \) and \( e_2 \triangleright u e \). We use the fact that \( \sigma \) is pure 
with \( e_2 \triangleright u e \) and \( \lambda_u(e) \in I_A \) to conclude that \( \lambda_u(e_2) \in I_B \). Then, from 
\( e_1 \triangleright u e_2 \) and \( \lambda_u(e_2) \in I_B \) we find that \( \lambda_u(e_1) \in I_C \) since \( \tau \) is pure. Finally, we use the definition of identity (Definition 5.11) on \( e' \triangleright u e_1 \) 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 \( \text{SynProc}_T \) 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 \( \text{eval}_{A,B} \) in \( \text{SynProc}_P \) with \( cs_{\text{Ck}} \). These are pure as tensoring does not 
alter justification pointers.

**Proposition 6.18.** \( \text{SynProc}' \) 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, \( \langle e_1, \text{tick} \rangle \cdot e_2 \cdot \langle e_2, \text{tick} \rangle \). 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 $\text{SynProc}'$ that synchronise with the global clock $Ck$. That is, those that can only receive inputs and produce outputs simultaneously with the clock reserved event $\text{tick}$. 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 $\text{CkProc}$ of clocked processes. Its objects are those of $\text{SynProc}'$. 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 $\lambda_s(e') = \text{tick}$. 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 $\sigma : A \rightarrow B$ 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 $\text{SynProc}'$.

For any signatures $A$, $B$ and $C$ we define the set of interaction traces, written $\text{int}_C(A, B, C)$ as $\Theta_C((A \Rightarrow B) \Rightarrow C)$.

**Definition 6.21 (Interaction of clocked processes).** The interaction $\sigma \otimes_C \tau$ of clocked processes $\sigma : A \rightarrow B$ and $\tau : B \rightarrow C$ is the set $\{u \in \text{int}_C(A, B, C) | u \upharpoonright A + B + Ck \in \sigma \text{ and } u \upharpoonright B + C + Ck \in \tau\}$.

**Definition 6.22 (Composition of clocked processes).** Let $\sigma : A \rightarrow B$ and $\tau : B \rightarrow C$ be clocked processes. Their composition $\sigma \circ_C \tau$ is the set $\{u \upharpoonright A + C + Ck | u \in \sigma \circ_C \tau\}$.

Intuitively, the clocked identity is the largest clocked process contained in $\eta_A$, where $\eta_A$ is the identity process in $\text{SynProc}_T$. It is defined as follows.

**Definition 6.23 (Clocked identity).** The identity morphism for object $A$, denoted by $\text{id}_A$, is the clocked process consisting of traces $u$ over $A_1 \Rightarrow A_2$ satisfying, for all events $e$ in $E_u \setminus \{e \in E_u | \lambda_u(e) \in \text{lin}(Ck)\}$, there exists an event $e' \neq e$, such that

1. $e \approx_u e'$ and
2. \[\lambda_u(e) = \text{inl}(a) \text{ if and only if } \lambda_u(e') = \text{inr}(a)\] and \[\lambda_u(e) = \text{inr}(a) \text{ if and only if } \lambda_u(e') = \text{inl}(a)\] and

3. (a) if \(\lambda_u(e) \in I_A\), then \(e' \sim_u e\)

(b) if \(\lambda_u(e) \in I_A\), then \(e \sim_u e'\)

(c) if \(\lambda_u(e) \not\in I_A\) and \(\lambda(e) \not\in I_A\), then there are \(e_1, e_2 \in E_u\) such that \(e_1 \sim_u e\) and \(e_2 \sim_u e'\) and \(e_1 \approx_u e_2\) and \([\lambda_u(e_1) = \text{inl}(a') \text{ if and only if } \lambda_u(e_2) = \text{inr}(a')]\) and \([\lambda_u(e_1) = \text{inr}(a') \text{ if and only if } \lambda_u(e_2) = \text{inl}(a')]\)

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

On objects, the tensor product is defined exactly as in \text{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 \(\{u \in \Theta_C(A \otimes C \Rightarrow B \otimes D) \mid u \upharpoonright A + B + Ck \in \sigma \text{ and } u \upharpoonright C + D + Ck \in \tau\}\).

**Lemma 6.25.** For clocked processes \(\sigma : A \to B\) and \(\tau : B \to C\), we have \(\sigma \otimes_C \tau = (\sigma \otimes T(\tau) \otimes \mu_C) \upharpoonright A + B + C + Ck\).

**Proof.** Showing that \(\sigma \otimes_C \tau \supseteq (\sigma \otimes T(\tau) \otimes \mu_C) \upharpoonright A + B + C + Ck\) is straightforward. So, we only prove that \(\sigma \otimes_C \tau \subseteq (\sigma \otimes T(\tau) \otimes \mu_C) \upharpoonright A + B + C + Ck\).

\[
\text{LHS} = \{v \in \Theta_C((A \Rightarrow B) \Rightarrow C \otimes Ck) \mid u \upharpoonright A + B + Ck \in \sigma \text{ and } u \upharpoonright B + C + Ck \in \tau\}
\]

\[
\text{RHS} = \{u \in \Theta_C(((A \Rightarrow B \otimes Ck_4) \Rightarrow (C' \otimes Ck_2) \otimes Ck_3) \Rightarrow C \otimes Ck) \mid u \upharpoonright A + B + Ck_4 \in \sigma \text{ and } u \upharpoonright B + C' + Ck_2 \in \tau \text{ and } u \upharpoonright Ck_4 + Ck_3 \in id_{Ck} \text{ and } u \upharpoonright C' + C \in id_C \text{ and } u \upharpoonright Ck_2 + Ck \in id_{Ck} \text{ and } u \upharpoonright Ck_3 + Ck \in id_{Ck}\}
\]

Take \(v \in \sigma \otimes_C \tau\). We construct a trace \(u \in \sigma \otimes T(\tau) \otimes \mu_C\) such that \(u \upharpoonright A + B + C + Ck = v\) as follows.

- \(E_u = E_v + E_v|Ck + E_v|C + Ck\).
- \(\lambda_v = \lambda_v + \lambda_v|Ck + \lambda_v|Ck + \lambda_v|C + Ck\).
• The preorder $\preceq_u$ is the relation satisfying the following, for all $e, e' \in E_v$.

P1 if $e, e' \in E_{v \upharpoonright A + B}$, then $in_1(e) \preceq_u in_1(e')$

P2 if $e \in E_{v \upharpoonright A + B}$ and $e' \in E_{v \upharpoonright C_k}$, then $e \preceq_v e' \iff in_1(e) \preceq_u in_1(e') \iff in_1(e) \preceq_u in_2(e') \iff in_1(e) \preceq u in_3(e') \iff in_1(e) \preceq u in_4(e')$

P3 if $e \in E_{v \upharpoonright A + B}$ and $e' \in E_{v \upharpoonright C}$, then $e \preceq_v e' \iff in_1(e) \preceq_u in_1(e') \iff in_1(e) \preceq u in_4(e')$

P4 if $e \in E_{v \upharpoonright C}$ and $e \in E_{v \upharpoonright A + B}$, then $e \preceq_v e' \iff in_1(e) \preceq_u in_1(e') \iff in_2(e) \preceq_u in_1(e') \iff in_3(e) \preceq_u in_1(e') \iff in_3(e) \preceq u in_4(e') \iff in_3(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_1(e')$

P5 if $e \in E_{v \upharpoonright C}$ and $e \in E_{v \upharpoonright A + B}$, then $e \preceq_v e' \iff in_1(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_1(e')$

P6 if $e \in E_{v \upharpoonright C}$ and $e \in E_{v \upharpoonright C}$, then $e \preceq_v e' \iff in_1(e) \preceq u in_1(e') \iff in_1(e) \preceq u in_4(e') \iff in_2(e) \preceq u in_1(e') \iff in_2(e) \preceq u in_4(e') \iff in_3(e) \preceq u in_1(e') \iff in_3(e) \preceq u in_4(e') \iff in_3(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_4(e')$

P7 if $e \in E_{v \upharpoonright C}$ and $e \in E_{v \upharpoonright C}$, then $e \preceq_v e' \iff in_1(e) \preceq u in_1(e') \iff in_1(e) \preceq u in_2(e') \iff in_1(e) \preceq u in_3(e') \iff in_1(e) \preceq u in_4(e') \iff in_1(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_4(e')$

P8 if $e, e' \in E_{v \upharpoonright C}$, then $e \preceq_v e' \iff in_1(e) \preceq u in_1(e') \iff in_1(e) \preceq u in_4(e') \iff in_1(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_4(e')$

P9 if $e, e' \in E_{v \upharpoonright C}$, then $e \preceq_v e' \iff in_1(e) \preceq u in_1(e') \iff in_1(e) \preceq u in_2(e') \iff in_1(e) \preceq u in_3(e') \iff in_2(e) \preceq u in_1(e') \iff in_2(e) \preceq u in_2(e') \iff in_2(e) \preceq u in_3(e') \iff in_3(e) \preceq u in_1(e') \iff in_3(e) \preceq u in_2(e') \iff in_3(e) \preceq u in_3(e') \iff in_4(e) \preceq u in_4(e') \iff in_4(e) \preceq u in_1(e') \iff in_4(e) \preceq u in_2(e') \iff in_4(e) \preceq u in_3(e') \iff in_4(e) \preceq u in_4(e')$

• Let $\bigtriangleup_u$ be defined as follows, for all $e, e' \in E_v$.

J1 If $e, e' \in E_{v \upharpoonright A + B}$, then $e \bigtriangleup_v e' \iff in_1(e) \bigtriangleup_u in_1(e')$.

J2 If $e, e' \in E_{v \upharpoonright C}$, then $e \bigtriangleup_v e' \iff in_1(e) \bigtriangleup_u in_1(e') \iff in_4(e) \bigtriangleup_u in_4(e')$, and $\lambda_v(e) \in I_C \iff in_4(e) \bigtriangleup_u in_1(e)$.

J3 If $e \in E_{v \upharpoonright C}$ and $e' \in E_{v \upharpoonright B}$, then $e \bigtriangleup_v e' \iff in_1(e) \bigtriangleup_u in_4(e) \iff in_4(e) \bigtriangleup_u in_1(e')$.

J4 If $e \in E_{v \upharpoonright C}$, then $in_1(e) \bigtriangleup_u in_2(e) \iff in_1(e) \bigtriangleup_u in_3(e) \iff in_3(e) \bigtriangleup_u in_4(e)$.

J5 The only pointers in $\bigtriangleup_u$ are those specified by (J1)–(J4); that is,
6.2. CLOCKED PROCESSES

- if $e \in E_{v|A}$ and $e' \in E_{v|B}$, then $in_1(e) \not\prec u in_1(e')$,
- if $e \in E_{v|A+B}$ and $e' \in E_{v|C}$, then $in_1(e) \not\prec u in_1(e')$ and $in_1(e) \not\prec u in_4(e')$,
- if $e \in E_{v|A+B}$ and $e' \in E_{v|Ck}$, then $in_1(e) \not\prec u in_1(e')$ and $in_1(e) \not\prec u in_2(e')$ and $in_1(e) \not\prec u in_3(e')$ and $in_1(e) \not\prec u in_4(e')$,
- if $e \in E_{v|C}$ and $e' \in E_{v|A}$, then $in_1(e) \not\prec u in_1(e')$ and $in_4(e) \not\prec u in_1(e')$,
- if $e \in E_{v|C}$ and $e' \in E_{v|B}$, then $in_1(e) \not\prec u in_1(e')$,
- if $e \in E_{v|C}$ and $e' \in E_{v|Ck}$, then $in_1(e) \not\prec u in_1(e')$ and $in_1(e) \not\prec u in_2(e')$ and $in_1(e) \not\prec u in_3(e')$ and $in_1(e) \not\prec u in_4(e')$ and $in_2(e')$ and $in_4(e) \not\prec u in_3(e')$ and $in_4(e) \not\prec u in_4(e')$,
- if $e, e' \in E_{v|Ck}$, then $in_1(e) \not\prec u in_1(e')$ and $in_1(e) \not\prec u in_4(e')$ and $in_2(e) \not\prec u in_1(e')$ and $in_2(e) \not\prec u in_2(e')$ and $in_2(e) \not\prec u in_3(e')$ and $in_2(e) \not\prec u in_4(e')$ and $in_3(e) \not\prec u in_1(e')$ and $in_3(e) \not\prec u in_2(e')$ and $in_3(e) \not\prec u in_3(e')$ and $in_3(e) \not\prec u in_4(e')$ and $in_4(e) \not\prec u in_1(e')$ and $in_4(e) \not\prec u in_3(e')$ and $in_4(e) \not\prec u in_4(e')$,
- if $e \in E_{v|Ck}$ and $e' \in E_{v|A+B}$, then $in_1(e) \not\prec u in_1(e')$ and $in_2(e) \not\prec u in_1(e')$ and $in_3(e) \not\prec u in_1(e')$ and $in_4(e) \not\prec u in_1(e')$,
- if $e \in E_{v|Ck}$ and $e' \in E_{v|C}$, then $in_1(e) \not\prec u in_1(e')$ and $in_2(e) \not\prec u in_1(e')$ and $in_3(e) \not\prec u in_1(e')$ and $in_4(e) \not\prec u in_1(e')$ and $in_1(e) \not\prec u in_4(e')$ and $in_2(e) \not\prec u in_4(e')$ and $in_3(e) \not\prec u in_4(e')$ and $in_4(e) \not\prec u in_4(e')$.

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.
Finally, we prove that $\lambda_v(e) \in L_C$ follows from (P8), (J2) and the fact that $\preceq_v$ is reflexive.

- $u \upharpoonright C' + C \in id_C$ follows from (P9), (J4) and the fact that $\preceq_v$ is reflexive.

- $u \upharpoonright Ck_4 + Ck_3 \in id_{Ck}$ follows from (P9), (J4) and the fact that $\preceq_v$ is reflexive.

- $u \upharpoonright Ck_2 + Ck \in id_{Ck}$ follows from (P9), (J4) and the fact that $\preceq_v$ is reflexive.

- $u \upharpoonright Ck_3 + Ck \in id_{Ck}$ follows from (P9), (J4) and the fact that $\preceq_v$ is reflexive.

- $u \upharpoonright A + B + Ck_4 \in \sigma$. We define a bijection $\phi : E_{u|A+B+Ck} \rightarrow E_{u|A+B+Ck_4}$ as follows.

  $$\phi(e) = \begin{cases} 
  in_1(e) & \text{if } e \in E_{u|A+B} \\
  in_4(e) & \text{if } e \in E_{u|Ck}
  \end{cases}$$

  The proof that the conditions in Definition 5.4 are met follows from the definitions of $E_u$ and $\lambda_u$; (P1); (P2); (P4); (P9); (J1) and (J4).

- $u \upharpoonright B + C' + Ck_2 \in \tau$. We define a bijection $\phi : E_{u|B+C+Ck} \rightarrow E_{u|B+C+Ck_2}$ as follows.

  $$\phi(e) = \begin{cases} 
  in_1(e) & \text{if } e \in E_{u|B} \\
  in_4(e) & \text{if } e \in E_{u|C} \\
  in_2(e) & \text{if } e \in E_{u|Ck_2}
  \end{cases}$$

  The proof that the conditions in Definition 5.4 are met follows from the definitions of $E_u$ and $\lambda_u$; (P3); (P4); (P5); (P6); (P7); (P9); (J1) and (J4).

Finally, we prove that $u \upharpoonright A + B + C + Ck = v$. From the definition of $u$ and the definition of projection, we see that $E_{u|A+B+C+Ck} = in_1(E_v)$ and $\lambda_{u|A+B+C+Ck} = \lambda_v \circ in_1$. We can also verify, by inspection of the definition of $\preceq_u$ that $(\forall e, e' \in E_v)(e \preceq_v e' \iff in_1(e) \preceq_{u|A+B+C+Ck} in_1(e'))$.

Next, we prove that for all $e, e' \in E_v$, we have $e \preceq_v e' \iff in_1(e) \preceq_{u|A+B+C+Ck} in_1(e')$ 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), $in_1(e) \preceq_{u} in_1(e')$ so $in_1(e) \preceq_{u|A+B+C+Ck} in_1(e')$.

- Case: $\lambda_v(e) \in L_B$ and $\lambda_v(e') \in L_B$. By (J1), $in_1(e) \preceq_{u} in_1(e')$ so $in_1(e) \preceq_{u|A+B+C+Ck} in_1(e')$. 

6.2. CLOCKED PROCESSES

- Case: \( \lambda_v(e) \in L_A \) and \( \lambda_v(e') \in L_A \). By (J1), \( \text{in}_1(e) \sim_u \text{in}_1(e') \) so \( \text{in}_1(e) \sim_{u|A+B+C+Ck} \text{in}_1(e') \).

- Case: \( \lambda_v(e) \in L_C \) and \( \lambda_v(e') \in L_B \). By (J3), \( \text{in}_1(e) \sim_u \text{in}_4(e) \) and \( \text{in}_4(e) \sim_u \text{in}_1(e') \) so \( \text{in}_1(e) \sim_{u|A+B+C+Ck} \text{in}_1(e') \).

- Case: \( \lambda_v(e) \in L_B \) and \( \lambda_v(e') \in L_A \). By (J1), \( \text{in}_1(e) \sim_u \text{in}_1(e') \) so \( \text{in}_1(e) \sim_{u|A+B+C+Ck} \text{in}_1(e') \).

- All other label assignments are illegal.

\[ \square \]

**Corollary 6.26.** For clocked processes \( \sigma : A \to B \) and \( \tau : B \to C \), we have \( \sigma \circ C \tau = \sigma \circ T \tau \).

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 \( \sigma \circ C \tau \) 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

\[ \text{Proof.} \] Let \( \sigma : A \to B \) be a clocked process. We have to prove that \( \sigma \circ C \text{id}_B = \sigma \). Since \( \text{id}_B \subseteq \mu_B \) and since all clocked processes are also processes in \( \text{SynProc}_r \), it is easy to see that \( \sigma \circ C \text{id}_B \subseteq \sigma \). We then prove that \( \sigma \circ C \text{id}_B \supseteq \sigma \). The proof follows the same structure as the proof of Lemma 3.24.

Let \( s \in \sigma \). We will find an interaction trace \( u \in \text{int}_C(A, B, B') \) such that \( u \upharpoonright A + B + Ck = s \) and \( u \upharpoonright B + B' + Ck \in \text{id}_B \) and \( u \upharpoonright A + B' + Ck = s \). Let \( u \) be defined as follows.

- \( E_u = E_s + E_{s|B} \).

- \( \lambda_u = \lambda_s + \lambda_{s|B} \).

- The preorder \( \preceq_u \) is defined as follows. For all \( e, e' \in E_s \),
CHAPTER 6. GLOBAL CLOCKS AND DETERMINISM

P1 if \( e, e' \in E_s \), then \( e \preceq_s e' \iff in_1(e) \preceq_u in_1(e') \)

P2 if \( e \in E_s \) and \( e' \in E_{s | B} \), then \( e \preceq_s e' \iff in_1(e) \preceq_u in_2(e') \)

P3 if \( e \in E_{s | B} \) and \( e' \in E_s \), then \( e \preceq_s e' \iff in_2(e) \preceq_u in_1(e') \)

P4 if \( e, e' \in E_{s | B} \), then \( e \preceq_s e' \iff in_1(e) \preceq_u in_1(e') \iff in_2(e) \preceq_u in_2(e') \iff in_1(e) \preceq_u in_2(e') \iff in_2(e') \preceq_u in_2(e') \)

- Define \( \bowtie_u \) as follows, for all \( e, e' \in E_s \).

  J1 If \( e, e' \in E_{s | A} \), then \( e \bowtie s e' \iff in_1(e) \bowtie_u in_1(e') \)

  J2 If \( e \in E_{s | B} \) and \( e' \in E_{s | A} \), then \( e \bowtie s e' \iff in_1(e) \bowtie_u in_1(e') \)

  J3 If \( e, e' \in E_{s | B} \), then \( e \bowtie s e' \iff in_1(e) \bowtie_u in_1(e') \iff in_2(e) \bowtie_u in_2(e') \)

  J4 \( \lambda_s(e) \in I_B \iff in_2(e) \bowtie_u in_1(e) \)

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

    - if \( e \in E_{s | A} \) and \( e' \in E_{s | B} \), then \([in_1(e) \not\bowtie_u in_1(e') \text{ and } in_1(e) \not\bowtie_u in_2(e')] \),
    - if \( e \in E_{s | B} \), then \( in_1(e) \not\bowtie_u in_2(e') \),
    - if \( e \in E_{s | B} \) and \( e' \in E_{s | A} \), then \( in_2(e) \not\bowtie_u in_1(e') \).

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 \((\forall e, e' \in E_u)(e \bowtie_u e' \Rightarrow e \preceq_u e')\) through a case study. Suppose \( e \bowtie_u e' \). There are three possibilities given by the definition of \( \bowtie_u \).

- \( in_1^{-1}(e) \bowtie_s in_1^{-1}(e') \) \( \therefore \) \( in_1^{-1}(e) \preceq_s in_1^{-1}(e') \) because \( s \) is justified \( \therefore e \preceq_u e' \) by (P1).

- \( in_2^{-1}(e) \bowtie_s in_2^{-1}(e') \) \( \therefore \) \( in_2^{-1}(e) \preceq_s in_2^{-1}(e') \) because \( s \) is justified \( \therefore e \preceq_u e' \) by (P4).

- \( in_2^{-1}(e) = in_1^{-1}(e') \) and \( in_2^{-1}(e), in_1^{-1}(e') \in E_{s | B} \) \( \therefore \) \( in_2^{-1}(e) \preceq_s in_1^{-1}(e') \) 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 \( e' \approx_s e \) where \( c \) is a clock event. We use (P1) to find that \( e \approx_u in_1(c) \).
6.2. CLOCKED PROCESSES

- \( \lambda_u(e) \in L_B \). So, \( e = in_2(e') \) and \( e' \approx_s e \) where \( e \) is a clock event. We use (P2) and (P3) to find that \( e \approx_u in_1(e) \).

- 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 \( u \upharpoonright A + B + Ck \) and \( u \upharpoonright B + B' + Ck \) and \( u \upharpoonright A + B' + Ck \) are clocked.

Because \( \preceq_s \) is reflexive, (P4) implies that \( (\forall e \in E_s|B)(in_1(e) \approx_u in_2(e)) \). We also have \( \lambda_u|B+B'\)(in_1(e)) = \lambda_u|B+B'(in_2(e)) = inr(b), b \in L_B \). Moreover, (J3) and (J4) ensure that pointers are assigned according to Definition 6.2.3. So, \( u \upharpoonright B + B' + Ck \in id_B \). We briefly argue that \( u \upharpoonright A + B = s \). We have \( E_u|A+B+Ck = in_1(e) \) and \( \lambda_u|A+B+Ck = \lambda_s \circ in_1 \). From the definition of \( \preceq_u \), we can see that \( (\forall e, e' \in E_s)(e \preceq_e e' \iff in_1(e) \approx_u|A+B+Ck in_1(e') \). We need to prove that \( (\forall e, e' \in E_s)(e \preceq e' \iff in_1(e) \approx_u|A+B+Ck in_1(e') \). This is clear from the definition of \( \preceq_u \).

We then show that \( u \upharpoonright A + B' + Ck = s \). We have \( E_u|A+B'+Ck = E_s|A+Ck + E_s|B \). There is a bijection \( \phi : E_s \rightarrow E_u|A+B+Ck \) defined as

\[
\phi(e) = \begin{cases} 
  in_1(e) & \text{if } e \in E_s|A+Ck \\
  in_2(e) & \text{if } e \in E_s|B 
\end{cases}
\]

It follows from the definition of \( u \) that \( \lambda_s|A+Ck = \lambda_s \circ \phi \). The proofs that \( (\forall e, e' \in E_s)(e \preceq_s e' \iff \phi(e) \preceq_u|A+B+Ck \phi(e') \) and \( (\forall e, e' \in E_s)(e \preceq_s e' \iff \phi(e) \preceq_u|A+B+Ck \phi(e') \) are by case study on the labels of \( e, e' \) and follow straightforwardly from the definition of \( u \).

The proof that \( u \upharpoonright A + B' = s \) 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 \( u \upharpoonright A + A' + Ck \in id_A \) and \( u \upharpoonright B + B' + Ck \in id_B \). Expanding this using Definition 6.2.3, we get
\begin{itemize}
\item $u \in \Theta_C(A \otimes B \Rightarrow A' \otimes B \otimes Ck)$ and
\item $(\forall e \in E_{u[A+A'+Ck]} \exists e' \in E_{u[A+A'+Ck]}(e \rightleftharpoons_{u} A+A'+Ck e')$ and
\item $(\forall e \in E_{u[A+A'+Ck]} \exists e' \in E_{u[A+A'+Ck]}(e \approx_{u} A+A'+Ck e' \text{ and } \lambda_{u[A+A'+Ck]}(e')) = \text{tick})$ and
\item $(\forall e \in E_{u[B+B'+Ck]} \exists e' \in E_{u[B+B'+Ck]}(e \rightleftharpoons_{u} B+B'+Ck e')$.
\item $(\forall e \in E_{u[B+B'+Ck]} \exists e' \in E_{u[B+B'+Ck]}(e \approx_{u} B+B'+Ck e' \text{ and } \lambda_{u[B+B'+Ck]}(e') = \text{tick})$.
\end{itemize}

Since $E_{u} = E_{u[A+A'+Ck]} \cup E_{u[B+B'+Ck]}$ and $(\forall e,e' \in E_{u})(\text{if } e \approx_{u} A+A'+Ck e' \text{ or } e \approx_{u} B+B'+Ck e',\text{ then } e \approx_{u} e')$ and for all $a \in E_{u[A+A'+Ck]}$, $b \in E_{u[B+B'+Ck]}$, we have $\lambda_{u[A+A'+Ck]}(a) = \lambda_{u}(a)$ and $\lambda_{u[B+B'+Ck]}(b) = \lambda_{u}(b)$, we conclude that for all $e \in E_{u}$ there is $e' \in E_{u}$ such that $e \rightleftharpoons_{u} e'$ and for all $e \in E_{u}$ there is $e' \in E_{u}$ such that $e \approx_{u} e'$ and $\lambda_{u}(e') = \text{tick}$. 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 $(\forall e \in E_{u} \exists e \in E_{u}(e \rightleftharpoons_{u} e'))$ and $(\forall e \in E_{u} \exists e' \in E_{u}(e \approx_{u} e' \text{ and } \lambda_{u}(e') = \text{tick})$. After projecting $u$ over $A + A' + Ck$, we get

\begin{itemize}
\item $u \upharpoonright A + A' + Ck \in \Theta_C(A \Rightarrow A' \otimes Ck)$ and
\item $(\forall e \in E_{u[A+A'+Ck]} \exists e' \in E_{u[A+A'+Ck]}(e \rightleftharpoons_{u} A+A' e')$ and
\item $(\forall e \in E_{u[A+A'+Ck]} \exists e' \in E_{u[A+A'+Ck]}(e \approx_{u} A+A'+Ck e' \text{ and } \lambda_{u[A+A'+Ck]}(e') = \text{tick})$.
\end{itemize}

that is, $u \upharpoonright A + A' + Ck \in id_{A}$. Projecting $u$ over $B + B' + Ck$ yields

\begin{itemize}
\item $u \upharpoonright B + B' + Ck \in \Theta_C(B \Rightarrow B' \otimes Ck)$ and
\item $(\forall e \in E_{u[B+B'+Ck]} \exists e' \in E_{u[B+B'+Ck]}(e \rightleftharpoons_{u} B+B' e')$ and
\item $(\forall e \in E_{u[B+B'+Ck]} \exists e' \in E_{u[B+B'+Ck]}(e \approx_{u} B+B'+Ck e' \text{ and } \lambda_{u[B+B'+Ck]}(e') = \text{tick})$.
\end{itemize}

that is, $u \upharpoonright B + B' \in id_{B}$. So, $u \in id_{A} \otimes id_{B}$. \hfill $\blacksquare$

**Lemma 6.32.** \textbf{CkProc} is a symmetric monoidal category.

**Proof.** In addition to Lemmas \textit{6.30} and \textit{6.31} we need to verify the following,
6.3. OBSERVABLE DETERMINISM

- **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 $\text{SynProc}'$, 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 $\text{SynProc}'$, 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.** $\text{CkProc}$ is a symmetric monoidal closed category.

The morphism $\text{eval}_{A,B}$ 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 $\text{final}(s)$, the set of final round events in a trace $s$, as follows.

$$\text{final}(s) = \{e \in E_s \mid (\forall e' \in E_s)(e' \preceq_s e)\}$$

We define $\text{final}^i(s), \text{final}^o(s)$ as the obvious restrictions to inputs and outputs, respectively. Let $\text{rest}(s)$ denote the trace $s$ less the last round. Formally, $\text{rest}(s)$ is defined by the following.

- $E_{\text{rest}(s)} = E_s \setminus \text{final}(s)$
- $\lambda_{\text{rest}(s)} = \lambda_s \mid E_{\text{rest}(s)}$
- $\preceq_{\text{rest}(s)} = \preceq_s \cap E_{\text{rest}(s)}^2$
• \( \eta_{rest(s)} = \eta_s \cap E_{rest(s)}^2 \)

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,

\[
\text{if } rest(s) = rest(s') \text{ and } final^l(s) = final^l(s'), \text{ then } final^o(s) = final^o(s') \quad (6.13)
\]

However, such definition yields an erroneous characterisation of determinism. Consider the following example.

**Example 6.34.** Let processes \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \), 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.

\[
\sigma = pc(\{ \langle b_1^1, a_1^1 \rangle, \langle b_2^2, a_2^2 \rangle \})
\]

\[
\tau = pc(\{ \langle c^1 \cdot a_1^1, b_1^2 \rangle, \langle c^1 \cdot a_2^2, b_2^2 \rangle \})
\]

According to (6.13), \( \sigma \) and \( \tau \) are deterministic. However, their composition \( \{ c^1 \cdot a_1^1, c^1 \cdot a_2^2 \} \) is not. The problem is that \( \tau \) 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 \( \phi : E_s \rightarrow E_{s'} \) is the trace equivalence bijection, we have for every output event \( o \) in final\( (s) \) and all inputs \( i \in E_s \) and \( i' \in E_{s'} \),

1. if \( i \not\approx_s o \) and \( i \not\approx_s o' \), then \( (\exists o' \in final(s'))(\lambda_s(o) = \lambda_{s'}(o') \text{ and } \phi(i) \not\approx_{s'} o') \)

2. if \( i \approx_s o \) and \( i \approx_{s'} o' \) and \( i' \in final(s') \) and \( \lambda_s(i) = \lambda_{s'}(i') \), then \( (\exists o' \in final(s'))(\lambda_s(o) = \lambda_{s'}(o') \text{ and } i' \not\approx_{s'} o') \)
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 $\text{tick}$. We can therefore use it as a counter. In each trace $v$, let $\text{tick}_n^v$ represent the $n^{th}$ clock event, where $n \in \mathbb{N}$, $[\text{tick}_n^v]$ the $n^{th}$ round, and $v_n$, the shortest prefix of $v$ containing $\text{tick}_n^v$. By convention, we let $v_0 = \epsilon$.

Lemma 6.36. For traces $s \in \sigma : A \rightarrow B$ and $t \in \tau : B \rightarrow C$, where $\sigma$ and $\tau$ are clocked processes, there is at most one interaction $u \in \sigma \land \tau$ 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, $s \downharpoonright t = \emptyset$
- 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 $[\text{tick}_n^s] \upharpoonright B + Ck = [\text{tick}_n^t] \upharpoonright B + Ck$, we have $[\text{tick}_n^u] = ([\text{tick}_n^s] \upharpoonright A + B + Ck) + ([\text{tick}_n^t] \upharpoonright C)$.

It follows that when $s$ and $t$ interact, they produce a single trace. 

For traces $s \in \sigma : A \rightarrow B$ and $\tau : B \rightarrow C$, where $\sigma$ and $\tau$ are clocked processes, if $s \downharpoonright t \neq \emptyset$, we say that $s$ and $t$ are composable. By slight abuse of notation, we refer to the unique trace in $s \downharpoonright t$ by $s \downharpoonright t$ and its projection over $A + C + Ck$ by $s; t$.

We now show that determinism is a compositional property in $Ck\text{Proc}$. 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 $\sigma : A \rightarrow B$ and $\tau : B \rightarrow C$ be deterministic processes in $Ck\text{Proc}$, with traces $s, s' \in \sigma$ and $t, t'$ in $\tau$. If $s$ and $t$ are composable and $s; t = s'; t'$, then $s = s'$ and $t = t'$. 

CHAPTER 6. GLOBAL CLOCKS AND DETERMINISM

Proof. By Definition 6.22 we have that \( s; t = s'; t' \) implies,

\[
(s\downarrow t) \uparrow A + C + Ck = (s'\downarrow t') \uparrow A + C + Ck
\]  
(6.14)

We prove that \( s\downarrow t = s'\downarrow t' \) by induction over the length of traces \( s, t, s', t' \) given by the number of rounds \( n \). Let \( u = s\downarrow t \) and \( u' = s'\downarrow t' \)

- Base case \( (n = 1) \). Suppose \( b \) is a \( B \)-event in \( \text{final}(u) \). Let \( b \) be justified by an event \( e \).

  - If \( b \) is initial in \( B \) then \( e \) is an input in \( \text{final}(t) \). By (6.14), \( e \in \text{final}(t') \). Since \( \tau \) is deterministic, \( b \in \text{final}(t') \).

  - 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 \( e_1 \leftarrow_u \cdots \leftarrow_u e_k \leftarrow_u b \) and \( e_1 \) is an initial \( B \)-event. We prove by induction that all events in \( Z \) are in \( \text{final}(u') \).

  * Base case \( (k = 1) \). We have \( e_1 \leftarrow_u b \). Then, there is a \( C \)-event \( c \in \text{final}(t) \) such that \( c \leftarrow \epsilon_i \). By (6.14), \( c \in \text{final}(t') \). Since \( \tau \) is deterministic, \( e_1 \in \text{final}(t') \). Since \( s' \) and \( t' \) compose, \( e_1 \in \text{final}(s') \). Since \( \sigma \) is deterministic, \( b \in \text{final}(s') \).

  * Inductive step. Suppose all events \( e_i \in Z \) where \( i \in \{1, \ldots, m\} \) are in \( \text{final}(u') \).

Now we consider the next event \( e_{m+1} \) in \( Z \). Suppose \( e_{m+1} \in \text{final}(u) \) and \( e_m \leftarrow_s e_{m+1} \). Let \( e_{m+1} \) be an output in \( \text{final}(s) \). By the inductive hypothesis, \( e_m \in \text{final}(s') \). Since \( \sigma \) is deterministic, we get \( e_{m+1} \in \text{final}(s') \). Let \( e_{m+1} \) be an input in \( \text{final}(s) \). Then, since \( s \) and \( t \) are composable, \( e_{m+1} \) is an output in \( \text{final}(t) \).

By the inductive hypothesis, \( e_m \in \text{final}(t') \). Since \( \tau \) is deterministic, we get \( e_{m+1} \in \text{final}(t') \).

- Inductive step. Suppose \( (s\downarrow t) \downarrow n = (s'\downarrow t') \downarrow n \). We show that rounds \( n + 1 \) are also equal in \( s\downarrow t \) and \( s'\downarrow t' \). Suppose \( b \) is a \( B \)-event in \( \text{final}(u) \). 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 \( \text{rest}(u) \). 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 \( b \in \text{final}(s') \) because \( \sigma \) is deterministic and if \( b \) is an output in \( t \) then \( b \in \text{final}(t') \) because \( \tau \) is deterministic.
We need to verify that \((\forall e, e' \in E_u)(e \sim_u e' \iff e \sim_{u'} e')\).

- If \(e\) is a \(C\)-event and \(e'\) is a \(C\)-event, then \(e \sim_{u'} e'\) from (6.14).
- If \(e\) is a \(A\)-event and \(e'\) is a \(A\)-event, then \(e \sim_{u'} e'\) 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 \(e \sim_{u'} e'\) because \(\sigma\) and \(\tau\) are deterministic.
- If \(e\) is a \(C\)-event and \(e'\) is a \(B\)-event, then \(e'\) is an output in \(t\). We have \(e \sim_{u'} e\) since \(\tau\) is deterministic.
- If \(e\) is a \(B\)-event and \(e'\) is a \(A\)-event, then \(e'\) is an output in \(s\). We have \(e \sim_{u'} e\) since \(\sigma\) is deterministic.

We can now show that deterministic processes compose.

**Lemma 6.38** (Compositionality). If \(\sigma : A \rightarrow B\) and \(\tau : B \rightarrow C\) are deterministic processes in \(C_{kProc}\), then \(\sigma; \tau\) is a deterministic process.

**Proof.** Let \(\sigma : A \rightarrow B\) and \(\tau : B \rightarrow C\) be deterministic processes in \(C_{kProc}\). By Definition 6.35 for all \(s, s' \in \sigma\) and \(t, t' \in \tau\) such that \(\text{rest}(s) = \text{rest}(s')\) and \(\text{rest}(t) = \text{rest}(t')\) we have,

\[
\begin{align*}
\text{if } i \sim_s o \text{ and } i \not\approx_s o, \text{ then } o & \in \text{final}(s') \quad (6.15) \\
\text{if } i \sim_s o \text{ and } i \approx_s o \text{ and } i \in \text{final}(s'), \text{ then } o & \in \text{final}(s') \quad (6.16) \\
\text{if } i \sim_t o \text{ and } i \not\approx_t o, \text{ then } o & \in \text{final}(t') \quad (6.17) \\
\text{if } i \sim_t o \text{ and } i \approx_t o \text{ and } i \in \text{final}(t'), \text{ then } o & \in \text{final}(t') \quad (6.18)
\end{align*}
\]

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 \(\text{rest}(v) = \text{rest}(v')\) and \(i \sim_v o\) and \(i \not\approx_v o\) and show that \(o \in \text{final}(v')\). Using Lemma 6.37 it follows that \(\text{rest}(s) = \text{rest}(s')\) and \(\text{rest}(t) = \text{rest}(t')\). From \(i \sim_v o\) 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, \( i \sim_v o \) implies \( i \sim_t o \). So we have \( i \sim_t o \) and \( i \not\approx_t o \). Using (6.17), we conclude that \( o \in \text{final}(t') \), and therefore \( o \in \text{final}(v') \).

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, \( i \sim_v o \) implies \( i \sim_s o \). So we have \( i \sim_s o \) and \( i \not\approx_s o \). Hence, using (6.15), we conclude that \( o \in \text{final}(s') \) and therefore, \( o \in \text{final}(v') \).

- 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, \( b \not\approx_s o \) and \( i \sim_t b \). 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 \( b \not\approx_s o \) and \( i \sim_t b \) and \( (i \not\approx_t b \) or \( b \not\approx_s o \)). This logically implies \( (b \not\approx_s o \) and \( b \not\approx_s o \)) or \( (i \sim_t b \) 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 \( o \in \text{final}(v') \).

For the second part, we assume that \( \text{rest}(v) = \text{rest}(v') \) and \( i \not\approx_v o \) and \( i \not\approx_v o \) and \( i \in \text{final}(v') \) and show that \( o \in \text{final}(v') \). Using Lemma 6.37 it follows that \( \text{rest}(s) = \text{rest}(s') \) and \( \text{rest}(t) = \text{rest}(t') \). From \( i \not\approx_v o \) and \( i \not\approx_v o \) and \( i \in \text{final}(v') \), 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 \( i \in \text{final}(v') \) that \( i \in \text{final}(t') \). So we have \( i \sim_v o \) and \( i \approx_t o \) and \( i \in \text{final}(t') \), and hence, using (6.18) we conclude that \( o \in \text{final}(t') \), and therefore \( o \in \text{final}(v') \).

- 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 \( i \in \text{final}(v') \) implies \( i \in \text{final}(s') \). So we have \( i \sim_s o \) and \( i \approx_s o \) and \( i \in \text{final}(s') \). Using (6.16) we conclude that \( o \in \text{final}(s') \), and therefore \( o \in \text{final}(v') \).
6.4. DISCUSSION

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, \( b \searrow_s o \) and \( i \searrow_t b \). 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 \( i \in \text{final}^l(v') \) which implies \( i \in \text{final}^l(t') \). Using (6.18), we deduce that \( b \in \text{final}^o(t') \).

This implies that \( b \in \text{final}^o(s') \) because \( s' \) and \( t' \) are assumed to be composable. Using (6.16), we deduce that \( o \in \text{final}(s') \) and consequently, \( o \in \text{final}(v') \).

We can now state the following result.

**Theorem 6.39.** There is a lluf subcategory of \( \text{CkProc} \) 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 $\sigma$ 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.

\[
\begin{align*}
s_1 &= \langle \text{tick }, a_1^t, a_1^o \rangle \cdot \langle a_2^t, a_2^o, \text{tick } \rangle \\
s_2 &= \langle \text{tick }, a_1^t, a_1^o \rangle \cdot \langle a_3^t, a_3^o, \text{tick } \rangle
\end{align*}
\]

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 $G$—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, JH90, 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—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 $\text{com}_1 \Rightarrow \text{com}_2 \times \text{com}_3$ and the following interpretation.

$$\begin{array}{c}
\text{com}_1 \Rightarrow \text{com}_2 \times \text{com}_3 \\
\quad r_1 \quad r_2 \\
\quad d_1 \quad d_2 \\
\quad r_3 \\
\quad r_2 \quad d_3
\end{array}$$

When activated ($r_3$), the strategy waits until $\text{com}_2$ is initiated, then terminates. Otherwise, the circuits behaves like the synchronous identity.
7.2. FURTHER DIRECTIONS

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


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


