# A Translation Measuring the Relative Expressiveness between CSP and ACP

*Leon Lee*

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2025

# Abstract

A popular technique for representing processes in Concurrency Theory is categorised into models called Process Algebras. Many Process Algebras exist, and efforts have been made to contextualise different Algebras egainst each other to find out notions of which Algebra are "better".

In this paper, we use the notion of "expressiveness", namely, "Can one Algebra express more tasks than another", and I present a translation between two popular Algebras, ACP and CSP, showing that ACP with Functional Renaming is at least as expressive as CSP up to rooted branching bisimulation.

# Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Leon Lee*)

# Acknowledgements

I would like to give my kindest regards to my supervisor Rob who always provided a patient and excellent guide on this journey into concurrency theory.

Thank you to all my friends for the late night study supports, and everybody at the Bedlam Theatre for always pushing me harder than should be possible.

Finally, thank you to my family for their constant support, as well as my lovely girlfriend Alice.

# Chapter 1

# Introduction

With the growing complexities of software and systems of the world, it is key to have methods of modelling more complex systems to get a better understanding of the underlying behaviour behind processes. Efforts have been made in sequential programming as early as the 1930's with Turing Machines and the $\lambda$-calculus. Systems in real life are rarely sequential, however, and usually involve multiple processes acting simultaneously, sometimes even synchronizing to interact with each other to perform tasks. These tasks that involve modelling multiple processes at once are referred to as a *concurrent system*.

It is clear to see that brute forcing solutions to these problems are significantly harder than a sequential system - the processing time will grow exponentially as the number of processes increase, and modelling a system like a colony of ants is near impossible. Therefore, we will need some way to formalise these concurrent systems. One category of models for concurrent systems are referred to as "Process Algebras", which will be the focus of this paper. These are languages similar to the Algebras of Mathematics, with processes built upon axioms, and often with familiar faces such as addition or multiplication but in slightly different contexts. This way of creation lets us compute results of complex models without the need for equally complex and multidimensional diagrams.

One of the big problems with Process Algebras is the ease of creation of new Algebras. The range of operators can vary language to language, with algebras existing for highly specific use-cases, since, why would you need to include an operator if you never use it? This leads to many different Process Algebras existing, and even multiple variants of a singular Process Algebra with slightly different tweaks added to it. To categorise all these different Algebras leads us to Expressiveness. The end goal of expressiveness is to create a hierarchy of different Process Algebras to see which algebras are more powerful. Put simply, if one Algebra can perform all tasks that another one can do, but not vice-versa, then it is clear that the first Algebra is more expressive.

This hierarchy of Expressiveness is vast and almost impossible to categorise into one paper, so I will be focusing on the expressiveness of two influencial algebras in the history of Concurrency - ACP and CSP. In **Chapter 2**, I will provide background information, a demonstration of a simple process algebra, and an introduction to our two algebras that we will focus on. A more formal definition of ACP and CSP will be defined in **Chapter 3**, as well as what it means for a language to be "more expressive" than another, and the method we will use to achieve it. In **Chapter 4**, I will define the actual translation between CSP and ACP, together with any relevant assistance that will be needed to help define it. In **Chapter 5**, I will provide a justification for the translation that I defined in the previous section, and finally in **Chapter 6**, I provide a summary of the results and conclusions of the paper.

# Chapter 2

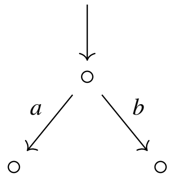
# Background

## 2.1 Process Algebra

Concurrency has been studied in many different ways, with the earliest attempts emerging from the 1960s, and some notable models being Petri nets, or the Actor Model. Process Algebras are one such method of modelling a Concurrent System, where the process is modelled in such a way that it is akin to the Universal Algebras of mathematics - in which operations are defined in an axiomatic approach to create a structurally sound way of defining processes in the system [3]. It is easily possible to model simple systems as a flow chart or diagram as you will be able to see throughout this paper, but a formal approach like process algebras will make way for modelling more complex systems, and lays the groundwork to provide a solid foundation to prove and base claims for such systems.

### 2.1.1 The Basic Process Algebra

A simple example in action is a process algebra where we only consider the alternative composition operator $+$, where applied to a process $a+b$ means "Choose $a$, or choose $b$". Processes are typically written in equations, but those equations can have a visual representation in the form of a **Process Graph**, which are diagrams that employ "states", and "actions" to show the traces, or paths, that a process can take. In this case, the process $a+b$ can be modelled in the following graph:




The graph begins at the top into the first node, and then can either progress to the left node via the action $a$, or the right node via the action $b$. If the process graph was a representation of an ATM, the graph could be thought of as the actions “View Money”, leading to a screen with a balance, or “Withdraw Money”, leading to a screen with different amounts to take out.

The axioms of the $+$ operator of BPA are as follows:

- **Commutativity**: $a+b$
- **Associativity**: $(a+b)+c = a+(b+c)$
- **Idempotency**: $a+a = a$

Comparable to the operation axioms of a Group or Ring in Mathematics, every other operation in a process algebra is constructed similarly. In practice, most process algebras will have some form of alternative composition, but this is a very simplified example and the developed algebras that exist are designed to handle a lot more complex situations, with the biggest feature being communication, which lets two processes act together at the same time with a “handshake” action.

### 2.1.2 Internal Actions

Process Algebras can feature an action that is unobservable, commonly referred to as $\tau$-actions, or the silent step. This is an action that isn’t decided by the user but rather an external source like a machine that is running the process. An example is the ATM, which might have a process

$$\bullet \xrightarrow{\text{Get User Value}} \bullet \xrightarrow{\text{Check Balance}} \bullet \xrightarrow{\text{Collect Notes}} \bullet \xrightarrow{\text{Open Dispenser}} \bullet \xrightarrow{\text{Dispense Money}} \bullet$$

but to an end user, this could just as well be the process

$$\bullet \xrightarrow{\text{Get User Value}} \bullet \xrightarrow{\tau} \bullet \xrightarrow{\text{Dispense Money}} \bullet$$

Where the machine-taken actions are simply abstracted into internal ones, or actions that cannot be affected by the user. Another example is a process featuring an internal choice, which is when an action is decided internally.

$$\begin{tikzcd}
& {} \arrow[d] & \\
& \circ \arrow[d, "\tau"] & \\
& \bullet \arrow[dl, "b"'] \arrow[dr, "c"] & \\
\bullet & & \bullet
\end{tikzcd}
\qquad
\begin{tikzcd}
& {} \arrow[d] & \\
& \circ \arrow[dl, "\tau"'] \arrow[dr, "\tau"] & \\
\bullet \arrow[d, "b"'] & & \bullet \arrow[d, "c"] \\
\bullet & & \bullet
\end{tikzcd}$$

In the left-hand side process, an internal action is made, followed by the user being able to pick between $b$ and $c$. However, in the right-hand side process, the choice is made **internally**, and the user will be locked out of the choice, being only able to pick one of the options.

### 2.1.3 Other Process Algebras

There are many process algebras that exist, the most famous and seminal being CSP [7], CCS [13], and ACP [5, 6], with some other popular calculi being the π-calculus and its various extensions [14, 16, 1] which have been used to varying degrees in fields like Biology, Business, and Cryptography, or the Ambient Calculus [8] which has been used to model mobile devices.

## 2.2 Encodings of Process Algebra

With the growing number of process algebras, one might begin to ask if there is a way of comparing different process algebras to each other to find the single best one, as a parallel to Turing Machines and the Church-Turing thesis. However, the wide range of applications that different process algebra are used for makes that rather impractical, and the goal of unifying all process algebra into a single theory seems further and further away as more process algebras for even more specified tasks get created.

A more reasonable approach is to compare different process algebras and their expressiveness, two main relevant methods being *absolute* and *relative* expressiveness.[17] Absolute expressiveness is the idea of comparing a specific process algebra to a question and seeing if it can solve the problem - e.g. if a process algebra is Turing Complete. However, this merely biparts different algebra - the process algebra that are able to solve a specified problem, and the ones who aren't [12]. Therefore, the question of relative expressiveness - i.e. how one language compares to another is a lot more useful in terms of categorising different process algebras by expressiveness.

A well studied way of comparing expressiveness is through an "encoding", and whether an algebra can be translated from one to another, but not vice versa [18]. The general notion of an encoding is not defined by clear boundaries, and the criterion for a valid encoding may vary from language to language, but work has been made to try and generalise the notion of a "valid" encoding [12, 22].

## 2.3 CSP

CSP (Communicating Sequential Processes) [7] is a Process Algebra developed by Tony Hoare based on the idea of message passing via communications. It was developed in the 1980s and was one of the first of its kind, alongside CCS by Milner. CSP uses the idea of action prefixing which is where operators are of the syntax $a \to P$, where $a$ is an event and $P$ is a process.

## 2.4 ACP

ACP (Algebra of Communicating Processes) [5] is a Process Algebra developed by Jan Bergstra and Jan Willem Klop. Compared to CSP, ACP is built up with an axiomatic approach in mind which does away with the idea of action prefixing and instead can allow for unguarded operations. $ACP_\tau$ [6] is an extension of ACP that includes the silent step $\tau$ described in Section 2.1.2

## 2.5 Comparing CSP to ACP

The biggest difference between CSP and ACP is the difference in Communication of parallel processes, as CSP uses conjugate actions to form communications, while ACP uses a separately defined communication function that can be defined over any action. An encoding between CSP and ACP has been described in [23]. In this paper, we extend this encoding to more CSP operations, and provide justifications for our encoding.

# Chapter 3

# A formal definition of CSP and $\mathrm{ACP}_F^\tau$

## 3.1 Languages and Expressiveness

We first define formally what it means to be a language.

**Definition 3.1: Languages**

Via [22], we can represent a language $\mathscr{L}$ as a pair $(\mathbb{T}, [\![\ ]\!])$, where $\mathbb{T}$ is a set of valid expressions in $\mathscr{L}$, and $[\![\ ]\!]$ is a mapping $[\![\ ]\!] : \mathbb{T} \to \mathscr{D}$ from $\mathbb{T}$ to a set of meanings $\mathscr{D}$. We also define $\mathscr{X}$, the set of process variables, such as specified in a recursive specification.

The expressiveness of two languages, $\mathscr{L}$ and $\mathscr{L}'$ can be measured using a Translation, i.e. a way to map expressions in one language to another

**Definition 3.2: Translation**

Via [22], a **translation** from a language $\mathscr{L}$ to a language $\mathscr{L}'$ is a mapping $\mathscr{T} : \mathbb{T}_{\mathscr{L}} \to \mathbb{T}_{\mathscr{L}'}$

There are two main notions of expressiveness, and our preferred way to measure expressiveness is via relative expressiveness rather than absolute expressiveness. Via [17], absolute expressiveness measures the way that processes compare against each other, i.e. if a process in an algebra can be represented by another. Relative expressiveness takes a more robust approach, in trying to encode the individual operations of the algebra compared to entire processes. From the encoding of operators, entire processes can then be constructed, therefore satisfying Absolute Expressiveness as well.

**Definition 3.3: Expressiveness**

Via [22], a language $\mathscr{L}'$ is **at least as expressive as** $\mathscr{L}$ iff a **valid** translation from $\mathscr{L}$ into $\mathscr{L}'$ exists.

The wording of "valid" is intentionally left vague, as there are many notions of validity. Validity is measured using a relation, and the strongest relation there is between two Algebas is a Bisimulation. This is a relation where any behaviour in an algebra can be identically replicated by another, therefore perfectly simulating each other. As we will see, while bisimulation is ideal, it is not always possible to achieve. We now define formally what it means to be a valid translation.

**Definition 3.4: Validity**

Via [22], we say that a translation $\mathscr{T} : \mathbb{T}_{\mathscr{L}} \to \mathbb{T}_{\mathscr{L}'}$ is **valid up to an equivalence** $\sim$ if we have that $\mathscr{T}(P) \sim P$, for all $P \in \mathbb{T}_{\mathscr{L}}$

Listed in [17] is also a range of weaker criterons that are desirable for a translation. One particular criteron that we will focus on is **compositionality**, which, when achieved, means that the translations of an operator is valid regardless of the context inside them, which means that any expression will be encodable from one algebra to another by translating smaller and smaller segments of the expression.

**Definition 3.5: Compositionality**

Via [22], a translation $\mathscr{T}$ from $\mathscr{L}$ into a language $\mathscr{L}'$ is **compositional** if $\mathscr{T}(X) = X$ for each $X \in \mathscr{X}$, and for each $n$-ary operator $f$ of $\mathscr{L}$ there exists an $n$'ary $\mathscr{L}$-context $C_f$ such that $\mathscr{T}(f(E_1,\ldots,E_n)) = C_f[\mathscr{T}(E_1,\ldots,\mathscr{T}(E_n))]$ for any $\mathscr{L}$ expressions $E_1,\ldots,E_n \in \mathbb{T}_{\mathscr{L}}$

We should also specify that $\mathscr{T}(a) = a$ for each $a \in \Sigma$, where $\Sigma$ is the set of actions, to make sure that translations of actions are consistent between algebras.

## 3.2 $\mathrm{ACP}^{\tau}_{F}$

### 3.2.1 Basic $\mathrm{ACP}_{\tau}$

**Definition 3.6:** $\mathrm{ACP}_{\tau}$

The process algebra $\mathrm{ACP}_{\tau}$ as described in [6] is parameterised by a set of actions, $\Sigma$, and a communication function $|$. The grammar of $\mathrm{ACP}_{\tau}$ can be described with the following operations:

$$P,Q ::= a \mid \delta \mid P.Q \mid P+Q \mid P \,||\, Q \mid P \,⌊⌊\, Q \mid P|Q \mid \partial_H(P) \mid \tau_I(P) \mid \tag{L1}$$

where the operators are: *action*, *deadlock*, *sequential composition*, *alternative composition*, *merge*, *left merge*, *communication merge*, *encapsulation*, and *abstraction*.

We have $H, I \subseteq \Sigma$, and additionally we define the set $\Sigma_\tau$ to be the set $\Sigma \cup \{\tau\}$, as the silent step is not included in $\Sigma$. We also write $\Rightarrow$ to indicate a chain of $\tau$-actions, possibly none. The operations of $\mathrm{ACP}_F^\tau$ can be described in the following manner:

tset

- **Action**, or $a$, is any action.
- **Deadlock**, or $\delta$, is the empty process. This can also be thought of as a process that does not terminate successfully.
- **Sequential Composition**, or $P.Q$ is an operation that performs $P$, and then performs $Q$.
- **Parallel Composition**, or $P+Q$ is a process that can perform $P$ or $Q$.
- **Restriction**, or $\partial_H(P)$, is a process with all actions in $H$ removed.
- **Abstraction**, or $\tau_I(P)$, is a process with all actions in $I$ renamed to internal actions, or $\tau$.

### 3.2.2 Communication and Merge

The operations Merge ($P \,||\, Q$), Left Merge ($P \,\lfloor\!\lfloor\, Q$), and Communication Merge ($P \mid Q$) form the basis of Communication in ACP. Compared to the other operators of ACP which symbolise actions *a or b*, and *a then b*, communication represents an action *a and b*, or in other words, a process that performs $a$ and $b$ simultaneously. The merge operation is characterised as

$$P \,||\, Q = P \lfloor\!\lfloor Q + Q \lfloor\!\lfloor P + P \mid Q$$

and along with a simplified axiom set of the Left Merge operator found in ACP [6],

$$a \lfloor\!\lfloor Q = a.Q \qquad a.P \lfloor\!\lfloor Q = a(P \,||\, Q)$$

The result of $P \,||\, Q$ is a lattice of any combination of moves of $P$, as well as any combination of moves of $Q$, while at each step also performing $P \mid Q$.

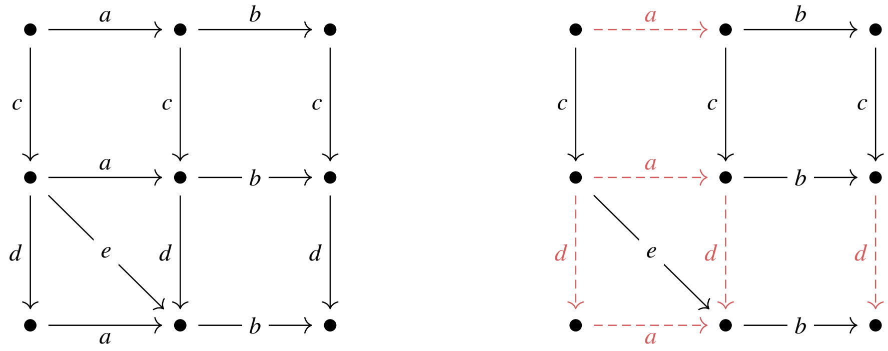
The operation $\mid$ is a function $A \times A \rightharpoonup A$ that defines valid communications between the two processes[1]. This can be thought of as a hand-shaking action between $P$ and $Q$. Together with the restriction operator, communications can now be formed between two processes. An example is shown in Figure 3.1.

### 3.2.3 Successful Termination

The language ACP is typically defined with a notion of Successful Termination, written $\checkmark$, where an identifier is added to states to signify that a process is finished, otherwise a state of deadlock, or $\delta$, is achieved. This results in twice as many Operational Rules, which we can see in the extended rule set of the language $\mathrm{ACP}_F^\tau$ shown in Table B.2, as well as different cases in any equivalences we will define.

This is quite a verbose way of describing the language, and there have been some alternatives proposed to avoid this. Two main examples are the extension $\mathrm{ACP}_\varepsilon$, which

[1] Communication Merge is a Partial Function, meaning it is not defined over all actions



(a) Communication and Left Merges (b) Restriction applied over the set $\{a,d\}$

Figure 3.1: Example of Communication on the process: $\partial_{\{a,d\}}(a.b||c.d)$. An example of how this can be modelled as a vending machine. Take the process $a.b$ to mean `accept payment.dispense item`, the process $c.d$ to mean `choose item.insert payment`, and the communication $a \mid d = e$ to mean `pay`. The process will then evaluate to the process `choose item.pay.dispense item`, with the process synchronising over `insert payment` and `recieve payment`.

adds a successful termination action at the cost of extending process graphs with more actions and the loss of a true Strong Bisimulation due to the extra action ε[24], and the fragment aprACP [21], which uses action prefixing similar to CSP or the language CCS at the cost of losing asymmetry and compatibility [11]. The language CSP does not distinguish successful termination from unsuccessful termination, and has only the process STOP, therefore for simplicity we will simply use ACP with standard termination, ignoring any irrelevant termination rules in equivalences and operation rules.

### 3.2.4 $\mathrm{ACP}_F^\tau$

A proposed extension of ACP adds a Functional Renaming operator, as shown in [21], which lets you rename actions via a function $f : A \times A \rightharpoonup A$. From this point forth, we will be using this extension, written as $ACP_F^\tau$. Our final grammar for the language $\mathrm{ACP}_F^\tau$ is formally defined below.

**Definition 3.7:** $\mathrm{ACP}_F^\tau$

The Algebra $\mathrm{ACP}_F^\tau$ as described in [6], [21], is parameterised by a set of actions, $\Sigma$, and a communication function $|$. The grammar of $\mathrm{ACP}_F^\tau$ can be described with the following operations:

$$P,Q ::= a \mid \delta \mid P.Q \mid P+Q \mid P||Q \mid P⌊⌊Q \mid P|Q \mid \partial_H(P) \mid \tau_I(P) \mid f(P) \mid \qquad \text{(L2)}$$

where the operators are: *action*, *deadlock*, *sequential composition*, *alternative composition*, *merge*, *left merge*, *communication merge*, *encapsulation*, *abstraction*, *functional renaming*.

## 3.3 CSP

**Definition 3.8:** *CSP*

The process algebra CSP as defined in [7, 15, 20] is parameterised on a set of communications $\Sigma$, and the grammar we will be using consists of the operations:

$$P,Q ::= \mathrm{STOP} \mid \mathrm{div} \mid a \to P \mid P \sqcap Q \mid P \,\square\, Q \mid P \rhd Q \mid \\ P \,||_A\, Q \mid P \backslash A \mid f(P) \mid P \triangle Q \mid P \,\Theta_A\, Q \mid \tag{L3}$$

where the operators are: *inaction*, *divergence*, *action prefixing*, *internal choice*, *external choice*, *sliding choice*, *parallel composition*, *concealment*, *renaming*, *interrupt*, and *throw*.

We have that $A \subseteq \Sigma$, and the operations can be described in the following manner:

- **Inaction**, or STOP, is the process that does nothing.
- **Divergence**, or div, is a process that is stuck in an infinite processing loop, and can never perform an external action. As explained in section 4.2, this can be thought as an infinite chain of internal actions.
- **Action Prefixing**, or $a \to P$, is an operation that performs the action $a$ followed by the process $P$. Note that $a$ must be a *single* action, therefore disallowing operations such as the process $(a \to b) \to (c \to d)$.
- **Internal Choice**, or $P \sqcap Q$, is an operation that can perform either $P$ or $Q$, but the choice is not decided by the user but rather by an outside decision, similar to flipping a coin.
- **External Choice**, or $P \,\square\, Q$, is an operation that can perform a choice of $P$ or $Q$, and the choice is decided by the user. Internal choices can still progress on each process and the External choice will not be satisfied[2]. This differs from ACP Alternative Composition, as in ACP, a $\tau$ action will satisfy the $+$ operation.
- **Sliding Choice**, or $P \rhd Q$, is an operation that acts like *external choice* on $P$, and *internal choice* on $Q$.
- **Parallel Composition**, or $P \,||_A\, Q$ is an operation that interleaves two processes together, similarly to ACP Merge. The difference being that in the language of CSP, the only actions that can synchronise are identical actions, i.e. the only action that can synchronise with an action $a$ in CSP is another $a$. On the other hand, in the language ACP, the actions that can synchronise with an action $a$ are any actions defined in the merge operator, $|$.
- **Concealment**, or $P \backslash Q$, is an operation that removes actions from a set $Q$, acting similarly to the Abstraction operator $\tau_I$ of ACP

[2]Here, we use "satisfied" to mean that the operator carries on after an action is executed

- **Renaming**, or $f(P)$, is an operation that renames actions in accordance to a function $f : A \times A \rightharpoonup A$.
- **Interrupt**, or $P \triangle Q$, is an operation that can perform visible actions of $P$, but the moment an action in $Q$ is made, it will then turn into $Q$ and stop other actions of $P$ from happening.
- **Throw**, or $P\,\Theta_A\,Q$, is an operation that has a set of "throw" actions, where the process can perform visible actions of $P$, but the moment an action in the "throw" set occurs, the process switches to $Q$. This can also be thought of as an error operator.

## 3.4 Structured Operational Semantics

Structured Operational Semantics [19] are a method of describing the actions of an operator. They are laid out in Proof tables, in the form:

$$\frac{\textbf{Statement A}}{\textbf{Statement B}} \qquad \frac{}{\textbf{Statement C}}$$

A double-sided proof tree means that **Statement A** implies **Statement B**, or can be read as "If **Statement A** is true, then so is **Statement B**". Proof tables can also be in the form of an Axiom, as shown in **Statement C**. This can be read as "if True is true, then so is **Statement C**", which is clearly valid.[3]

To show an example of an operation in the SOS Format, we will show the rules of ACP Merge as explained in Section 3.2.2:

$$\frac{P \xrightarrow{\alpha} P'}{P\,||\,Q \xrightarrow{\alpha} P'\,||\,Q} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q' \quad a\,|\,b = c}{P\,||\,Q \xrightarrow{a} P'\,||\,Q'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P\,||\,Q \xrightarrow{\alpha} P\,||\,Q'}$$

where the meaning of the proof tables is:

- If $P$ can perform an action $\alpha$ to $P'$, then $P\,||\,Q$ can perform an action $\alpha$ to $P'\,||\,Q$.
- If $P$ can perform an action $a$ to $P'$, and $Q$ can perform an action $b$ to $Q'$ such that $a|b = c$ is defined on the Communication Merge, then $P\,||\,Q$ can perform an action $c$ to $P'\,||\,Q'$.
- If $Q$ can perform an action $\alpha$ to $Q'$, then $P\,||\,Q$ can perform an action $\alpha$ to $P\,||\,Q'$.

The rest of the operations of $\text{ACP}_F^\tau$ and CSP can also be represented similarly in Tables 3.1 and 3.2:

[3] We will express axioms without the Proof Table line for clarity

| | | | |
|---|---|---|---|
| $a \xrightarrow{\alpha} \checkmark$ | $\dfrac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$ | $\dfrac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$ | $\dfrac{P \xrightarrow{\alpha} \checkmark}{P \cdot Q \xrightarrow{\alpha} Q}$ |
| $\dfrac{P \xrightarrow{\alpha} P'}{P \cdot Q \xrightarrow{\alpha} P' \cdot Q}$ | $\dfrac{P \xrightarrow{\alpha} P'}{P \lfloor\!\lfloor Q \xrightarrow{\alpha} P' \,\|\, Q}$ | $\dfrac{P \xrightarrow{\alpha} P'}{P \,\|\, Q \xrightarrow{\alpha} P' \,\|\, Q}$ | $\dfrac{Q \xrightarrow{\alpha} Q'}{P \,\|\, Q \xrightarrow{\alpha} P \,\|\, Q'}$ |
| $\dfrac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{b} Q' \quad a\|b=c}{P \,\|\, Q \xrightarrow{c} P' \,\|\, Q'}$ | $\dfrac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{b} Q' \quad a\|b=c}{P \mid Q \xrightarrow{c} P' \,\|\, Q'}$ | | $\dfrac{P \xrightarrow{\alpha} P' \quad (\alpha \in I)}{\tau_I(P) \xrightarrow{\tau} \tau_I(P')}$ |
| $\dfrac{P \xrightarrow{\alpha} P' \quad (\alpha \notin I)}{\tau_I(P) \xrightarrow{\alpha} \tau_I(P')}$ | $\dfrac{P \xrightarrow{\alpha} P' \quad (\alpha \notin H)}{\partial_H(P) \xrightarrow{\alpha} \partial_H(P')}$ | $\dfrac{\langle \mathcal{S}_X \mid \mathcal{S} \rangle \xrightarrow{\alpha} P'}{\langle X \mid \mathcal{S} \rangle \xrightarrow{\alpha} P'}$ | $\dfrac{P \xrightarrow{\alpha} P'}{f(P) \xrightarrow{f(a)} f(P')}$ |

Table 3.1: Structural operational semantics of the language $\mathrm{ACP}_F^\tau$. The operational semantics for Successful Termination in operations other than Sequential Composition are omitted. A full list of operational rules can be found in Table B.2

| | | | |
|---|---|---|---|
| $\mathrm{div} \xrightarrow{\tau} \mathrm{div}$ | $(a \to P) \xrightarrow{a} P$ | $P \sqcap Q \xrightarrow{\tau} P$ | $P \sqcap Q \xrightarrow{\tau} Q$ |
| $\dfrac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'}$ | $\dfrac{P \xrightarrow{\tau} P'}{P \Box Q \xrightarrow{\tau} P' \Box Q}$ | $\dfrac{Q \xrightarrow{a} Q'}{P \Box Q \xrightarrow{a} Q'}$ | $\dfrac{Q \xrightarrow{\tau} Q'}{P \Box Q \xrightarrow{\tau} P \Box Q'}$ |
| $\dfrac{P \xrightarrow{a} P'}{P \triangleright Q \xrightarrow{a} P'}$ | $\dfrac{P \xrightarrow{\tau} P'}{P \triangleright Q \xrightarrow{\tau} P' \triangleright Q}$ | $P \triangleright Q \xrightarrow{\tau} Q$ | $\dfrac{P \xrightarrow{\alpha} P'}{f(P) \xrightarrow{f(\alpha)} f(P')}$ |
| $\dfrac{P \xrightarrow{\alpha} P' \quad (\alpha \notin A)}{P \|_A Q \xrightarrow{\alpha} P' \|_A Q}$ | $\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q' \quad (a \in A)}{P \|_A Q \xrightarrow{a} P' \|_A Q'}$ | | $\dfrac{Q \xrightarrow{\alpha} Q' \quad (\alpha \notin A)}{P \|_A Q \xrightarrow{\alpha} P \|_A Q'}$ |
| $\dfrac{P \xrightarrow{\alpha} P' \quad (\alpha \notin A)}{P \backslash A \xrightarrow{\alpha} P' \backslash A}$ | $\dfrac{P \xrightarrow{a} P' \quad (a \in A)}{P \backslash A \xrightarrow{\tau} P' \backslash A}$ | $\dfrac{P \xrightarrow{\alpha} P' \quad (\alpha \notin A)}{P \,\Theta_A\, Q \xrightarrow{a} P' \,\Theta_A\, Q}$ | $\dfrac{P \xrightarrow{a} P' \quad (a \in A)}{P \,\Theta_A\, Q \xrightarrow{a} Q}$ |
| $\dfrac{P \xrightarrow{\alpha} P'}{P \triangle Q \xrightarrow{\alpha} P' \triangle Q}$ | $\dfrac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P' \triangle Q'}$ | $\dfrac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} Q'}$ | $\mu p.P \xrightarrow{\tau} P[\mu p.P/p]$ |

Table 3.2: Structural operational semantics of CSP

## 3.5 Semantic Equivalences

Two notions of equivalence that we will be focusing on is strong bisimilarity, and rooted branching bisimulation. As said before, strong bisimilarity is the finest equivalence one can have between translations. We define formally the definition of strong bisimulation via [4]:

**Definition 3.9: Strong Bisimulation**

Let $P$ and $Q$ be two processes, and $R$ be a relation between nodes of $P$ and nodes of $Q$. $R$ is a **Strong bisimulation** between $P$ and $Q$ if:

1. The roots of $P$ and $Q$ are related by $R$.
2. If $s \xrightarrow{\alpha} s'$ for $\alpha \in \Sigma_\tau$ is an edge in $P$, and $sRt$, then there is an edge $t \xrightarrow{\alpha} t'$ such that $s'Rt'$.
3. If $t \xrightarrow{\alpha} t'$ for $\alpha \in \Sigma_\tau$ is an edge in $P$, and $sRt$, then there is an edge $s \xrightarrow{\alpha} s'$ such that $s'Rt'$.
4. If $sRt$, then $s\checkmark$ iff $t\checkmark$.[a]

[a] As explained in Section 3.2.3, this rule can be omitted without a loss in expressiveness over CSP.

To show the difference between different notions of equivalence, we will compare *Bisimulation Semantics* to *Trace Semantics*. **Trace Semantics** looks at different *traces*, or *paths* that a process can take. For example, in a process $P$, where $P$ is defined in the language $\text{ACP}_F^\tau$ as $a.(b+c)$, we can look at *Completed Trace Equivalence*. This is an equivalence that says that two processes are equivalent if they have the same set of Completed Traces[4]. The traces of $P$ are therefore defined as the set

$$\{a.b,\, a.c\}$$

In Trace Semantics, the process $P$ is equivalent to the process $Q := a.b + a.c$, since they both have the traces $\{a.b,\, a.c\}$. However, this is clearly not the case in Bisimulation Semantics, where rule 2 in Definition 3.9 is not satisfied, as there is no state in $Q$ that can perform both a $b$ action, and a $c$ action. Clearly, this implies that *Bisimulation equivalence* is a **finer** relation than *Completed Trace Equivalence*. Here, *finer* means that one equivalence can distinguish between more processes than another. We will also use **coarser** to mean the opposite.

Moving on from Trace Semantics, we will look at Branching bisimilarity, a relation where processes are deemed branching bisimilar (or BB) if they can move the same

[4] Here, **Completed Trace** refers to a path the process can take from start to end

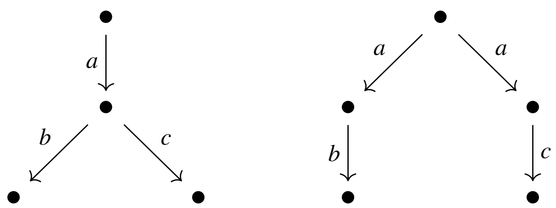



Figure 3.2: The process $a.(b+c)$ (left), compared to the process $a.b+a.c$ (right). Both Processes have the same Traces - $\{a.b,\, a.c\}$, however they are not strongly bisimilar

.

way when looking at **External actions**. In particular, this excludes internal actions, $\tau$, by saying that two processes are equivalent if two states are related, even when there are some number of internal actions between them.

A finer relation than branching bisimilarity is Rooted branching bisimilarity (RBB), which adds the condition that the first action in the process must be strongly bisimilar. This is a preferred relation to regular branching bisimilarity since Rooted branching bisimilarity is a Congruence [10]. This means that RBB is not only finer than BB, but also a more robust relation, since regular branching bisimilarity isn't compositional on operations such as ACP $+$. We define formally the definition of branching bisimulation and Rooted branching bisimulation via [4]:

**Definition 3.10: Rooted branching bisimilarity**

Let $P$ and $Q$ be two processes, and $R$ be a relation between nodes of $P$ and nodes of $Q$. $R$ is a **Branching bisimulation** between $P$ and $Q$ if:

1. The roots of $P$ and $Q$ are related by $R$.
2. If $s \xrightarrow{\alpha} s'$ for $\alpha \in \Sigma_\tau$ is an edge in $P$, and $sRt$, then either:
   a) $\alpha = \tau$ and $s'Rt$.
   b) $\exists t \Rightarrow t_1 \xrightarrow{\alpha} t'$ such that $sRt_1$ and $sRt'$.[a]
3. If $s\checkmark$ and $sRt$ then there exists a $t \Rightarrow t'$ in $Q$ to a state $t'$ with $t'\checkmark$ and $sRt'$[b].

4, 5 : As in $2, 3$, with the roles of $P$ and $Q$ interchanged.

$R$ is called a **Rooted branching bisimulation** if the following root condition is also satisfied:

- If $\mathrm{root}(P) \xrightarrow{\alpha} s'$ for $\alpha \in \Sigma_\tau$, then there is a $t'$ with $\mathrm{root}(Q) \xrightarrow{\alpha} t'$ and $s'Rt'$.
- If $\mathrm{root}(Q) \xrightarrow{\alpha} t'$ for $\alpha \in \Sigma_\tau$, then there is an $s'$ with $\mathrm{root}(P) \xrightarrow{\alpha} s'$ and $s'Rt'$.
- $\mathrm{root}(P)\checkmark$ iff $\mathrm{root}(Q)\checkmark$[b].

[a]Here, we use the symbol $\Rightarrow$ to mean a chain of $\tau$ actions, possibly none.

[b]As explained in Section 3.2.3, this rule can be omitted without loss in function over the translation

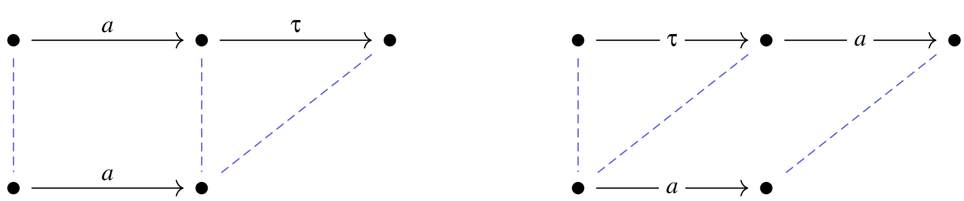



Figure 3.3: Example of branching bisimilarity. The processes on the left ($a.\tau$ compared to $a$) are rooted branching bisimilar, and the processes on the right ($\tau.a$ compared to $a$) are only branching bisimilar. A blue dotted line indicates a relation between states.

# Chapter 4

# A Translation of CSP to $\mathrm{ACP}_F^\tau$

## 4.1 Direct Translations

Some of the basic operations of CSP have an equivalent counterpart in ACP, with the only difference being the syntax. These can be easily translated in the following table:

$$\mathscr{T}(STOP) = \delta \qquad \text{(T1)} \qquad\qquad \mathscr{T}(a \to P) = a.\mathscr{T}(P) \qquad \text{(T2)}$$
$$\mathscr{T}(P \backslash A) = \tau_A(\mathscr{T}(P)) \qquad \text{(T3)} \qquad\qquad \mathscr{T}(\mu X.P) = \langle X \mid X = \tau.\mathscr{T}(P)\rangle \quad \text{(T4)}$$

## 4.2 Trivial Translations

- **Divergence** is the process that diverges via infinite internal actions, implying that a user can never make a decision past that point. It is defined by the rule div $\xrightarrow{\tau}$ div, and can be directly translated using recursion into $\mathrm{ACP}_F^\tau$ as the following equation:
$$\mathscr{T}(\mathrm{div}) = \langle X \mid X = \tau.X\rangle. \tag{T5}$$
- **Renaming** is an operation that renames actions in processes according to a function. There is no equivalent operator in plain $\mathrm{ACP}_\tau$, with the closest operation being $\tau_I(P)$ which abstracts actions in a set $I$ to internal actions. This is however possible in the extension $\mathrm{ACP}_F^\tau$, and in fact our translation is trivially
$$\mathscr{T}(f(P)) = f(\mathscr{T}(P)). \tag{T6}$$
- **Internal Choice** is an operation that emulates a choice of actions that isn't decided by the user, for example the result of flipping a virtual coin. CSP differentiates between external and internal choice, while in ACP the alternative choice operator $+$ handles both options, with internal actions satisfying[1] the $+$ operator. Along with the internal action $\tau$, a translation for CSP internal choice into $\mathrm{ACP}_F^\tau$ is written as
$$\mathscr{T}(P \sqcap Q) = \tau.\mathscr{T}(P) + \tau.\mathscr{T}(Q). \tag{T7}$$

The above translations are all valid up to Strong Bisimilarity.

[1]Here, we use **satisfying** to mean the resulting equation does not include the preceding operation.

## 4.3 Helper Operators for the language $\mathrm{ACP}_F^\tau$

### 4.3.1 Subsets of $\Sigma$

Working in $\mathrm{ACP}_F^\tau$ as defined in Definition 3.2, we first recall that $\mathrm{ACP}_F^\tau$ is parametrised by a set of actions $\Sigma$. Also recall that $\Sigma_\tau$ is defined as the set $\Sigma \cup \{\tau\}$. We start by defining some subsets of $\Sigma$ which we will use in our encodings.

**Definition 4.1: Subsets of A**

The set $\Sigma \in \mathbb{T}_{\mathrm{ACP}_F^\tau}$ is the set of all possible actions.

- $\Sigma_0 \subseteq \Sigma$ is the set of actions that get used in processes.
- $A \subseteq \Sigma_0$ is a set of target actions. This is used in operators such as CSP Parallel Composition (4.4.2), which communicates over a specified set.
- $H_0 = \Sigma - \Sigma_0$ is the set of working space operators, or in other words, any action that doesn't get used in processes.
- $\mathscr{H} \subseteq H_0$ is a selectively chosen set from $H_0$ to aid a translation.
- $H_1 = \Sigma_0 \uplus \mathscr{H}$ is the set of actions, plus any actions of $\mathscr{H}$. This can also be thought of the set of actions used in a translation.

In general, we have that $A \subseteq \Sigma_0 \subseteq H_1 \subseteq \Sigma$, and $\mathscr{H} \subseteq H_0 \subseteq \Sigma$.

We also have $\Sigma_0 \cap H_0 = \emptyset$, and by extension, $\Sigma_0 \cap \mathscr{H} = \emptyset$.

### 4.3.2 Triggering

We define an operator $\Gamma(P)$ that emulates the Triggering operator of MEIJE [2, 9]. For traces $a.b.\ldots$ of a process $P$, triggering can be represented as an operator that tags the first action of said traces. First, we define a function $f_{\mathtt{trig}}$, and communications for two actions, "`first`" and "`next`", that are defined in the working space $H_0$.

**Definition 4.2: Communications for the Triggering Operator**

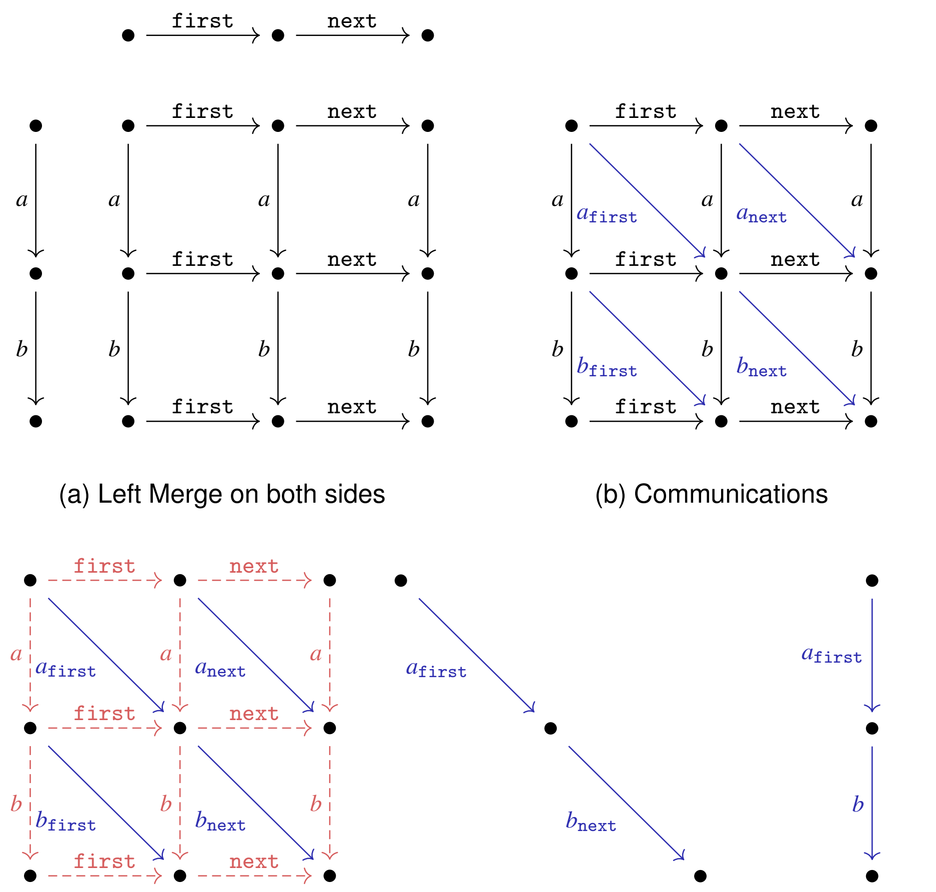
Define communications where:

$$a|\mathtt{first} = a_{\mathtt{first}}, \quad a|\mathtt{next} = a_{\mathtt{next}}. \tag{C1}$$

**Def 4.3: Triggering Function**

Define $f_{\mathtt{trig}} : \Sigma_\tau \to \Sigma_\tau$ where:

$$f_{\mathtt{trig}}(\alpha) = \begin{cases} a_{\mathtt{ini}} & \text{if } \alpha = a_{\mathtt{first}} \\ a & \text{if } \alpha = a_{\mathtt{next}} \\ \alpha & \text{otherwise} \end{cases} \tag{F1}$$




(a) Left Merge on both sides (b) Communications

(c) Restriction of $H_1$ (d) Result of Restriction (e) $f_{\texttt{trig}}$ applied

Figure 4.1: Example of $\Gamma(P)$ applied to $P = a.b.c$

We use the notation of $a^\infty$ as syntactic sugar to mean $\langle X \mid X = a.X \rangle$. Using the sets defined in Definition 4.1, we can now define $\Gamma(P)$.

**Definition 4.4: Triggering in ACP**

The **Triggering** operator is defined as the following process equation:

$$\Gamma(P) := f_{\texttt{trig}}\left[\partial_{H_1}(P \,||\, \texttt{first}.(\texttt{next}^\infty))\right] \tag{O1}$$

$\Gamma(P)$ is an operator that turns each trace $a.b.c.\dots$ of a process $P$ into the trace

$$a_{\texttt{ini}}.b.c.\cdots. \tag{4.1}$$

This works in the following method:

a) Merge the process $P$ with the process $\texttt{first}.(\texttt{next}^\infty)$. Via Definition 4.2, this will produce a lattice of $P$ and $\texttt{first.next.next}\dots$, with communications for every pair, but most importantly, a chain of communications going through the center of the graph of the form

$$a_{\mathtt{first}}.b_{\mathtt{next}}.c_{\mathtt{next}}.\cdots. \tag{4.2}$$

b) Restrict the actions in $H_1$, as defined in Definition 4.1, using the operator $\partial_{H_1}$. Since all the actions in both the processes $P$ and $\mathtt{first}.(\mathtt{next}^\infty)$ are in the set $H_1$, this effectively restricts both sides of the left merge, leaving only communications from the initial state. This leaves Equation 4.2 as the only remaining trace.

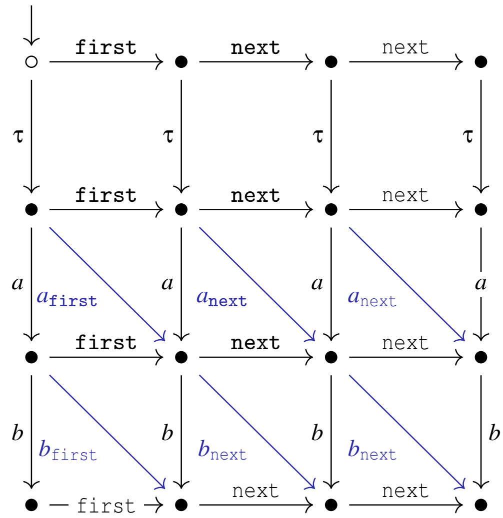
c) Apply $f_{\mathtt{trig}}$ as defined in Equation F1 to Equation 4.2, renaming the equation to

$$a_{\mathtt{ini}}.b.c.\cdots. \tag{4.3}$$

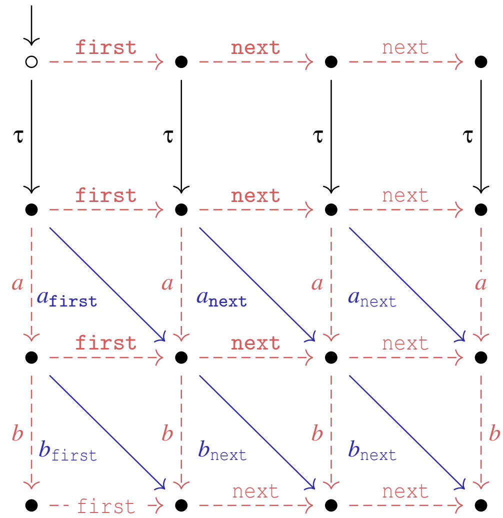
The process is now exactly as stated in Definition 4.4.

Note that since $\tau \notin \Sigma$, the restriction operator $\partial_{H_1}$ will therefore not affect any $\tau$ actions. Additionally, since $\tau$ does not communicate with any actions, Step b) effectively means "*some amount of* $\tau$ *steps, followed by the diagonal trace immediately following*". This results in traces such as the one shown below in Equation 4.4, where the operator effectively skips the $\tau$ action, then acts the same as if the process didn't start with a $\tau$.

$$\Gamma(\tau.b.c) = \tau.b_{\mathtt{ini}}.c \tag{4.4}$$




(a) Left merge with $\tau$ communications




(b) Restriction of $H_1$ with $\tau$ actions

Figure 4.2: Example of Triggering operator with internal actions

### 4.3.3 Associativity and Postfix Function

Shown in Table 4.1 is a list taken from the axioms of $\mathrm{ACP}_\tau$ [6] of the rules associated with the communication operator, $|$. Bearing these axioms in mind, we have the potential to run into problems with our communications, especially with the associativity rule

$$(a \mid b) \mid c = a \mid (b \mid c). \tag{4.5}$$

| **Communication function in** $\text{ACP}_\tau$ |
|---|
| $a\|b = b\|a$ |
| $(a\|b)\|c = a\|(b\|c)$ |
| $\delta\|a = \delta$ |

Table 4.1: Axioms of $\text{ACP}_\tau$ Communication

For example, in our proposed translation for the CSP External Choice operator, as shown in Section 4.4.3, a simplified version of the translation could have the following communications:

$$a|\texttt{first} = a_{\texttt{first}}, \qquad a|\texttt{next} = a_{\texttt{next}}, \qquad a_{\texttt{first}}|\texttt{choose} = a. \tag{4.6}$$

The Associativity axiom does not hold true, such as in the following counterexample:

$$a|\texttt{first}|\texttt{choose} = (a|\texttt{first})|\texttt{choose} = a_{\texttt{first}}|\texttt{choose} = a \tag{4.7}$$

$$a|\texttt{first}|\texttt{choose} = a|(\texttt{first}|\texttt{choose}) = a|\delta = \delta \tag{4.8}$$

Since Equation 4.7 is not equal to Equation 4.8, the associativity rule stated in Equation 4.5 does not hold true. This is also the reason that Definition 4.3 features the rule $f_{\texttt{trig}}(a_{\texttt{first}}) = a_{\texttt{ini}}$ instead of $f_{\texttt{trig}}(a_{\texttt{first}}) = a_{\texttt{first}}$. A preferred list of communications for external choice, and the one demonstrated in Section 4.4.3 is as follows:

$$a|\texttt{first} = a_{\texttt{first}} \qquad a|\texttt{next} = a_{\texttt{next}} \qquad a_{\texttt{ini}}|\texttt{choose} = a \tag{4.9}$$

From these communications, associativity of the communications now hold.

$$a|\texttt{first}|\texttt{choose} = (a|\texttt{first})|\texttt{choose} = a_{\texttt{first}}|\texttt{choose} = \delta \tag{4.10}$$

$$a|\texttt{first}|\texttt{choose} = a|(\texttt{first}|\texttt{choose}) = a|\delta = \delta \tag{4.11}$$

It is important to note that $\Gamma(P)$ takes precedence and hence the communication $\texttt{first}|\texttt{choose}$ would never occur for our application of External Choice, but the communication function must work over every action regardless of whether it will get used in practice for it to be correct for the axioms of ACP. To fix this, we define a compatibility function, $f_{\texttt{post}}$. We tag any potential renamings with $a_{\texttt{post}}$ to act as a filler for communications. Now, our final step will be to rename $a_{post}$ back to $a$ for any affected actions. This works in the following way:

$$a \xrightarrow{\text{Rename for Communication}} a_{\texttt{tag}} \xrightarrow{\text{Communicate with an action}} a_{\texttt{post}} \xrightarrow{\text{Rename back}} a \tag{4.12}$$

**Definition 4.5: Postfix function**

Let $f_{\text{post}} : \Sigma_\tau \to \Sigma_\tau$ be defined by the following rules:

$$f_{\texttt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\text{post}} \\ \alpha & \text{otherwise} \end{cases} \tag{F2}$$

## 4.4 Translations for the remaining CSP Operators

### 4.4.1 Communications and Functional Renaming

We define functions for our translation in addition to the ones previously defined in Sections 4.3.2 and 4.3.3. Note that these are defined over $\Sigma_\tau$ for bookkeeping purposes, as internal actions cannot get renamed and therefore stay as a $\tau$ no matter the function.

**Definition 4.6: Helper Functions**

Recall the functions $f_{\texttt{trig}}$ and $f_{\texttt{postfix}}$ defined in Equations F1 and F2:

$$f_{\texttt{trig}}(\alpha) = \begin{cases} a_{\texttt{ini}} & \text{if } \alpha = a_{\texttt{first}} \\ a & \text{if } \alpha = a_{\texttt{next}} \\ \alpha & \text{otherwise} \end{cases} \qquad f_{\texttt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\text{post}} \\ \alpha & \text{otherwise} \end{cases} \tag{F1, F2}$$

We define functions for the remaining operators below. We use the notation $A_T$ to signify a target set, as used in Sections 4.4.2 and 4.4.6.

- $f_{\text{syn}} : \Sigma_\tau \to \Sigma_\tau$ is a function that renames any actions in the target set $A$. This is used in the translation of Parallel Composition (4.4.2).

$$f_{\text{syn}}(\alpha) = \begin{cases} \alpha_{\texttt{syn}} & \text{if } \alpha \in A \\ \alpha & \text{otherwise} \end{cases} \tag{F3}$$

- $f_{\texttt{origin}} : \Sigma_\tau \to \Sigma_\tau$ is a function that renames actions in a process for use in operators. This is used in the translation of the Interrupt operator (4.4.5).

$$f_{\texttt{origin}}(\alpha) = \begin{cases} a_{\texttt{origin}} & \text{if } a \in \Sigma \\ \tau & \text{otherwise} \end{cases} \tag{F4}$$

- $f_{\texttt{split}} : \Sigma_\tau \to \Sigma_\tau$ is a function that renames actions in a process for use in operators, and also renames actions in the target set $A$. This is used in the translation of the Throw operator (4.4.6).

$$f_{\texttt{split}}(\alpha) = \begin{cases} a_{\texttt{split}} & \text{if } a \in A \\ a_{\texttt{origin}} & \text{if } a \notin A,\, a \in \Sigma \\ \tau & \text{otherwise} \end{cases} \tag{F5}$$

To aid these functions, we also define communications to use in our translation in addition to the ones previously defined in Section 4.3.2.

**Definition 4.7: Communications**

Recall the communications for the Triggering operator defined in Equation C1:

$$a|\texttt{first} = a_{\texttt{first}}, \qquad a|\texttt{next} = a_{\texttt{next}}. \tag{C1}$$

We define our additional communications. These all communicate to $a_{\texttt{post}}$, as motivated by Section 4.3.3.

- Communication for the $a_{\texttt{syn}}$ tag. This is used in the translation of Parallel Composition (4.4.2).
$$a_{\texttt{syn}}|a_{\texttt{syn}} = a_{\texttt{post}} \tag{C2}$$
- Communication for the $a_{\texttt{ini}}$ tag. This is used in the translation of External Choice (4.4.3), and Sliding Choice (4.4.4)
$$a_{\texttt{ini}}|\texttt{choose} = a_{\texttt{post}} \tag{C3}$$
- Communication for the $a_{\texttt{origin}}$ tag. This is used in the translation of the Interrupt and Throw operator (4.4.5, 4.4.6).
$$a_{\texttt{origin}}|\texttt{origin} = a_{\texttt{post}} \tag{C4}$$
- Communications for the $a_{\texttt{split}}$ tag. Equation C5 is used in the translation of the Interrupt operator (4.4.5), and Equation C6 is used the translation of the Throw operator (4.4.6).
$$a_{\texttt{ini}}|\texttt{split} = a_{\texttt{post}} \tag{C5}$$
$$a_{\texttt{split}}|\texttt{split} = a_{\texttt{post}} \tag{C6}$$

### 4.4.2 Parallel Composition

Parallel composition, $||_A$, is defined with the following rules from Table 3.2:

$$\frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin A)}{P\,||_A\,Q \xrightarrow{\alpha} P'\,||_A\,Q} \qquad \frac{Q \xrightarrow{\alpha} Q' \quad (\alpha \notin A)}{P\,||_A\,Q \xrightarrow{a} P\,||_A\,Q'} \qquad \frac{P \xrightarrow{a} P'\; Q \xrightarrow{a} Q' (\alpha \in A)}{P\,||_A\,Q \xrightarrow{a} P'\,||_A\,Q'} \tag{S1}$$

This operator functions mostly the same as the $\mathrm{ACP}_F^\tau$ counterpart, Merge, as explained in Section 3.2.2. The one difference is that in CSP, the action must be the same in $P$ and $Q$ to communicate, whereas in $\mathrm{ACP}_F^\tau$ communications are defined with a function, $|$.

For our encoding, we take $\mathscr{H} = \{\}$, and therefore $H_1 = \Sigma_0$. The goal is to tag actions in the target set $A$, and then define a communication function between identically marked actions. We can do this using the following functions and communications:

**Definition 4.8: Functions and Communications - Parallel Composition**

As defined in Definitions 4.6 and 4.7, we use Equations F2, F3 and C2.

$$f_{\mathtt{syn}}(\alpha) = \begin{cases} a_{\mathtt{syn}} & \text{if } \alpha \in A \\ \alpha & \text{otherwise} \end{cases} \qquad f_{\mathtt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\text{post}} \\ \alpha & \text{otherwise} \end{cases} \tag{F2, F3}$$

$$a_{\mathtt{syn}} | a_{\mathtt{syn}} = a_{\mathtt{post}} \tag{C2}$$

We now define our encoding of CSP Parallel Composition $||_A$ as the following equation:

$$\mathscr{T}(P \,||_A\, Q) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P)) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big) \tag{T8}$$

This translation is valid up to Rooted Branching Bisimilarity, shown in 5.2.1, and examples of the translation are shown in Appendix A.1.5.

### 4.4.3 External choice

The external choice operator $\Box$ is defined with the following rules from Table 3.2:

$$\frac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P \Box Q \xrightarrow{a} Q'} \qquad \frac{P \xrightarrow{\tau} P'}{P \Box Q \xrightarrow{\tau} P' \Box Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \Box Q \xrightarrow{a} P \Box Q'} \tag{S2}$$

In other words, we can take an external choice by the user, and additionally an internal action will still let an external choice be made after the internal move has been made. This differs from the $\text{ACP}_F^\tau$ Alternative Choice operator $(+)$, as $+$ will not let you select externally if an internal action is made.

For our encoding, we take $\mathscr{H} = \{\mathtt{first}, \mathtt{next}, \mathtt{choose}\}$, and therefore we have that $H_1 = A_0 \uplus \{\mathtt{first}, \mathtt{next}, \mathtt{choose}\}$ as defined in Definition 4.1.

**Definition 4.9: Functions and Communications - External Choice**

As defined in Definitions 4.6 and 4.7, we use Equations F1, F2 and C3.

$$f_{\mathtt{trig}}(\alpha) = \begin{cases} a_{\mathtt{ini}} & \text{if } \alpha = a_{\mathtt{first}} \\ a & \text{if } \alpha = a_{\mathtt{next}} \\ \alpha & \text{otherwise} \end{cases} \qquad f_{\mathtt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\text{post}} \\ \alpha & \text{otherwise} \end{cases}$$

$$a_{\mathtt{ini}} | \mathtt{choose} = a_{\mathtt{post}}$$

Additionally, recall the Triggering operator as defined in O1:

$$\Gamma(P) := f_{\mathtt{trig}}[\partial_{H_1}(P \,||\, \mathtt{first}(\mathtt{next}^\infty))]$$

We now define our encoding of CSP External Choice $\Box$ as the following equation:

$$\mathscr{T}(P \Box Q) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[\Gamma[\mathscr{T}(P)] \,||\, \mathtt{choose} \,||\, \Gamma[\mathscr{T}(Q)]\Big]\Big) \tag{T9}$$

This translation is valid up to Rooted Branching Bisimilarity, shown in Section 5.2.2, and examples of the translation are shown in Appendix A.1.1.

### 4.4.4 Sliding Choice

The Sliding Choice operator $\triangleright$ is defined with the following rules from Table 3.2:

$$\frac{P \xrightarrow{a} P'}{P \triangleright Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{\tau} P'}{P \triangleright Q \xrightarrow{\tau} P' \triangleright Q} \qquad P \triangleright Q \xrightarrow{\tau} Q \tag{S7}$$

In other words, this operator lets you take an external action on $P$. However, at any point before the external action, $P$ may "time out" and internally move to $Q$ instead.

For our encoding, we take $\mathscr{H} = \{\mathtt{first}, \mathtt{next}, \mathtt{shift}\}$, and therefore we have that $H_1 = \Sigma_0 \uplus \{\mathtt{first}, \mathtt{next}, \mathtt{shift}\}$ as defined in Definition 4.1. This translation is valid up to Rooted Branching Bisimilarity, shown in Section 5.2.3, and examples of the translation are shown in Appendix A.1.2.

**Definition 4.10: Functions and Communications - Sliding Choice**

Similarly to External Choice, we use Equations F1, F2 and C3 as defined in Definitions 4.6 and 4.7.

$$f_{\mathtt{trig}}(\alpha) = \begin{cases} a_{\mathtt{ini}} & \text{if } \alpha = a_{\mathtt{first}} \\ a & \text{if } \alpha = a_{\mathtt{next}} \\ \alpha & \text{otherwise} \end{cases} \qquad f_{\mathtt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\mathtt{post}} \\ \alpha & \text{otherwise} \end{cases} \tag{F1, F2}$$

$$a_{\mathtt{ini}} | \mathtt{choose} = a_{\mathtt{post}} \tag{C3}$$

Additionally, recall the Triggering operator as defined in O1:

$$\Gamma(P) := f_{\mathtt{trig}}[\partial_{H_1}(P \,||\, \mathtt{first}(\mathtt{next}^\infty))] \tag{O1}$$

We now define our encoding of CSP Sliding Choice $\triangleright$ as the following equation:

$$\mathscr{T}(P \triangleright Q) = \tau_{\{\mathtt{shift}\}}\Big(\partial_{H_0}\Big(f_{\mathtt{post}}\Big[\Gamma(\mathscr{T}(P)) \,||\, \mathtt{choose} \,||\, \mathtt{shift}_{\mathtt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big) \tag{T10}$$

### 4.4.5 Interrupt

The Interrupt operator $\triangle$ is defined with the following rules from Table 3.2:

$$\frac{P \xrightarrow{\alpha} P'}{P \triangle Q \xrightarrow{\alpha} P' \triangle Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'} \qquad \frac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} Q'}$$

In other words, we can take an external action from $P$ without satisfying[2] the operator,

[2] Here, we use "satisfies" to mean that the operator carries on after an action is executed

in addition to internal actions from $Q$. However, the moment an external action is made from $Q$, the process can then never return to $P$.

For our encoding, we take $\mathscr{H} = \{\texttt{first}, \texttt{next}, \texttt{origin}, \texttt{split}\}$, and therefore we have that $H_1 = \Sigma_0 \uplus \{\texttt{first}, \texttt{next}, \texttt{origin}, \texttt{split}\}$ as defined in Definition 4.1.

**Definition 4.11: Functions and Communications - Interrupt**

As defined in Definitions 4.6 and 4.7, we use Equations F1, F2, F4, C4 and C5.

$$f_{\texttt{trig}}(\alpha) = \begin{cases} a_{\texttt{ini}} & \text{if } \alpha = a_{\texttt{first}} \\ a & \text{if } \alpha = a_{\texttt{next}} \\ \alpha & \text{otherwise} \end{cases} \qquad f_{\texttt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\texttt{post}} \\ \alpha & \text{otherwise} \end{cases}$$

$$f_{\texttt{origin}}(\alpha) = \begin{cases} a_{\texttt{origin}} & \text{if } a \in \Sigma \\ \tau & \text{otherwise} \end{cases} \qquad \begin{aligned} a_{\texttt{origin}} | \texttt{origin} &= a_{\texttt{post}} \\ a_{\texttt{ini}} | \texttt{split} &= a_{\texttt{post}} \end{aligned}$$

Additionally, recall the Triggering operator as defined in O1:

$$\Gamma(P) := f_{\texttt{trig}}[\partial_{H_1}(P \,||\, \texttt{first}(\texttt{next}^{\infty}))])$$

We can now define our encoding of the CSP Interrupt operator $\triangle$ in $\mathrm{ACP}_F^\tau$. We start off with a new helper process, which we will call $\Pi$. This is defined as the recursive equation

$$\Pi = \langle X \mid X = \texttt{origin}.X + \texttt{split} \rangle$$

Or, visualised as a process graph:

$$\begin{tikzcd} {} \arrow[r] & \circ \arrow[loop above, "\texttt{origin}"] \arrow[r, "\texttt{split}"] & \bullet \end{tikzcd} \tag{O2}$$

We now define our encoding of CSP Interrupt $\triangle$ as the following equation:

$$\mathscr{T}(P \triangle Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P)) \,||\, \Pi \,||\, \Gamma(\mathscr{T}(Q))\Big]\Big) \tag{T11}$$

### 4.4.6 Throw

The Throw operator $\Theta_A$ is defined with the following rules:

$$\frac{P \xrightarrow{\alpha} P' \quad (a \notin A)}{P \,\Theta_A\, Q \xrightarrow{\alpha} P' \,\Theta_A\, Q} \qquad \frac{P \xrightarrow{a} P' \quad (a \in A)}{P \,\Theta_A\, Q \xrightarrow{a} Q}$$

In other words, we can take as many actions in $P$ as we want as long as they aren't contained in a set of actions, $A$. However, the moment an action in $A$ is made, the process then diverts to $Q$. This can be also thought as an error catching operator, with a set of "error" actions that switches to an "exception" process.

Similarly to Interrupt, for our encoding we take $\mathscr{H} = \{\mathtt{first}, \mathtt{next}, \mathtt{origin}, \mathtt{split}\}$. Therefore, we have $H_1 = \Sigma_0 \uplus \{\mathtt{first}, \mathtt{next}, \mathtt{origin}, \mathtt{split}\}$ as defined in 4.1.

**Definition 4.12: Functions and Communications - Throw**

As defined in Definitions 4.6 and 4.7, we use Equations F2, F5, C4 and C6.

$$f_{\mathtt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\text{post}} \\ \alpha & \text{otherwise} \end{cases} \qquad f_{\mathtt{split}}(\alpha) = \begin{cases} a_{\mathtt{split}} & \text{if } a \in A \\ a_{\mathtt{origin}} & \text{if } a \notin A,\, a \in \Sigma \\ \tau & \text{otherwise} \end{cases}$$

$$a_{\mathtt{origin}} | \mathtt{origin} = a_{\mathtt{post}} \qquad a_{\mathtt{split}} | \mathtt{split} = a_{\mathtt{post}}$$

We can now define our encoding of the CSP Throw operator $\Theta_A$. We employ the use of the same helper process $\Pi$, defined in the translation of the Interrupt operator O2.

$$\Pi = \langle X \mid X = \mathtt{origin}.X + \mathtt{split} \rangle$$

From this, our encoding can be written as the following equation:

$$\mathscr{T}(P \,\Theta_A\, Q) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{split}}(\mathscr{T}(P)) || \Pi.\mathscr{T}(Q)\Big]\Big) \tag{T12}$$

### 4.4.7 Final Translation

Recall the grammar of the languages CSP as shown in Definitions 3.8:

$$\begin{gathered} P, Q ::= \mathrm{STOP} \mid \mathrm{div} \mid a \to P \mid P \sqcap Q \mid P \,\Box\, Q \mid P \triangleright Q \mid \\ P ||_A Q \mid P \backslash A \mid f(P) \mid P \triangle Q \mid P \theta_A Q \mid \end{gathered}$$

Recall also the grammar of the language $\mathrm{ACP}_F^\tau$ as shown in Definition 3.7:

$$P, Q ::= a \mid \delta \mid P.Q \mid P + Q \mid P || Q \mid P \mathbin{|\!\lfloor} Q \mid P | Q \mid \partial_H(P) \mid \tau_I(P) \mid f(P) \mid$$

Finally, recall from Definitions 4.6 and 4.7 the list of functions and communications:

- $f_{\mathtt{trig}}(\alpha) = \begin{cases} a_{\mathtt{ini}} & \text{if } \alpha = a_{\mathtt{first}} \\ a & \text{if } \alpha = a_{\mathtt{next}} \\ \alpha & \text{otherwise} \end{cases}$
- $f_{\mathtt{origin}}(\alpha) = \begin{cases} a_{\mathtt{origin}} & \text{if } a \in \Sigma \\ \tau & \text{otherwise} \end{cases}$
- $f_{\mathtt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\text{post}} \\ \alpha & \text{otherwise} \end{cases}$
- $f_{\mathtt{split}}(\alpha) = \begin{cases} a_{\mathtt{split}} & \text{if } a \in A \\ a_{\mathtt{origin}} & \text{if } a \notin A,\, a \in \Sigma \\ \tau & \text{otherwise} \end{cases}$
- $f_{\mathtt{syn}}(\alpha) = \begin{cases} \alpha_{\mathtt{syn}} & \text{if } \alpha \in A \\ \alpha & \text{otherwise} \end{cases}$

- $a|\texttt{first} = a_{\texttt{first}}$
  $a|\texttt{next} = a_{\texttt{next}}$
- $a_{\texttt{ini}}|\texttt{choose} = a_{\texttt{post}}$
- $a_{\texttt{ini}}|\texttt{split} = a_{\texttt{post}}$
- $a_{\texttt{syn}}|a_{\texttt{syn}} = a_{\texttt{post}}$
- $a_{\texttt{origin}}|\texttt{origin} = a_{\texttt{post}}$
- $a_{\texttt{split}}|\texttt{split} = a_{\texttt{post}}$

We take $\mathscr{H} = \{\texttt{first}, \texttt{next}, \texttt{origin}, \texttt{split}, \texttt{shift}, \texttt{choose}\}$, and therefore we have that $H_1 = A_0 \uplus \{\texttt{first}, \texttt{next}, \texttt{origin}, \texttt{split}\}$ as defined in 4.1. We can now define our full translation:

**Definition 4.13: Translation of CSP to ACP**

Let $\mathbb{T}_{\mathrm{CSP}}$ be the expressions in the language CSP, and $\mathbb{T}_{\mathrm{ACP}_F^\tau}$ be expressions in the language $\mathrm{ACP}_F^\tau$. We define a translation $\mathscr{T} : \mathbb{T}_{\mathrm{CSP}} \to \mathbb{T}_{\mathrm{ACP}_F^\tau}$ defined as such:

$$\mathscr{T}(STOP) = \delta$$
$$\mathscr{T}(\mathrm{div}) = \langle X \mid X = \tau.X \rangle$$
$$\mathscr{T}(a \to P) = a.\mathscr{T}(P)$$
$$\mathscr{T}(P \sqcap Q) = \tau.\mathscr{T}(P) + \tau.\mathscr{T}(Q)$$
$$\mathscr{T}(P \Box Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma[\mathscr{T}(P)] \mid\mid \texttt{choose} \mid\mid \Gamma[\mathscr{T}(Q)]\Big]\Big)$$
$$\mathscr{T}(P \triangleright Q) = \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(P)) \mid\mid \texttt{choose} \mid\mid \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big)$$
$$\mathscr{T}(P \mid\mid_A Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{syn}}(\mathscr{T}(P)) \mid\mid f_{\texttt{syn}}(\mathscr{T}(Q))\Big]\Big)$$
$$\mathscr{T}(P \backslash A) = \partial_A \mathscr{T}(P)$$
$$\mathscr{T}(f(P)) = f(\mathscr{T}(P))$$
$$\mathscr{T}(P \triangle Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P)) \mid\mid \Pi \mid\mid \Gamma(\mathscr{T}(Q))\Big]\Big)$$
$$\mathscr{T}(P \Theta_A Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P)) \mid\mid \Pi.\mathscr{T}(Q)\Big]\Big)$$

# Chapter 5

# Validity of the Encoding

Referring back to our translation of External choice shown in Section 4.4.3:

$$\partial_{H_0}(f_{\mathtt{post}}[\Gamma(P)\,||\,\mathtt{choose}\,||\,\Gamma(Q)]) \tag{T9}$$

The translation has identical behaviour to $P+Q$ when translating processes with only external actions. However, on processes with $\tau$ actions, the translation is not so trivial. The addition of an internal action in the grammar of $\mathrm{ACP}^{\tau}_{F}$ works in our translation's favour for actions such as deferring an `ini` tag to the first visible action (see Figure 4.2). However, our translation largely relies on removing unwanted left-merges and communications through restriction operators. As internal actions are not able to interact with other operators, this results in restrictions that *should* remove all unneccessary actions ending up with unwanted left-over $\tau$ moves.
For example, if we take the process $a\Box\tau.b \in$ CSP, the resulting translation in $\mathrm{ACP}^{\tau}_{F}$ is

$$\partial_{H_0}\Big(f_{\mathtt{post}}\Big[\Gamma(a)\,||\,\mathtt{choose}\,||\,\Gamma(\tau.b)\Big]\Big) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[a_{\mathtt{ini}}\,||\,\mathtt{choose}\,||\,\tau.b_{\mathtt{ini}}\Big]\Big), \tag{5.1}$$

which has the following process graph:

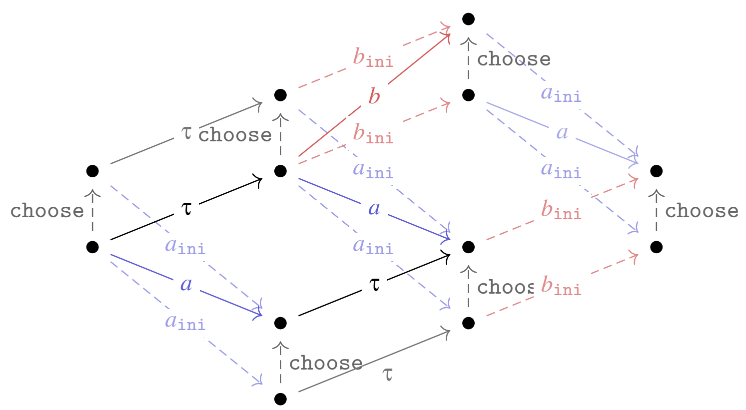



Figure 5.1: Counterexample for strong bisimilarity for the processes $P\Box\tau.b$. The result of the translation is $a.\tau+\tau.(a+b) \not\leftrightarrow a+\tau.(a+b)$

## 5.1 Prerequisites and Helper Theorems

### 5.1.1 Rooted Branching bisimilarity

Recall from Definition 3.10 the definition of branching bisimilarity and rooted branching bisimilarity:

**Recall 3.10: Branching Bisimulation**

A relation $R$ is a **Branching bisimulation**[a] between $P$ and $Q$ if:

1. The roots of $P$ and $Q$ are related by $R$.
2. If $s \xrightarrow{\alpha} s'$ for $\alpha \in \Sigma_\tau$ is an edge in $P$, and $sRt$, then either:
   a) $\alpha = \tau$ and $s'Rt$.
   b) $\exists t \Rightarrow t_1 \xrightarrow{\alpha} t'$ such that $sRt_1$ and $sRt'$.[b]
4. If $t \xrightarrow{\alpha} t'$ for $\alpha \in \Sigma_\tau$ is an edge in $Q$, and $sRt$, then either:
   a) $\alpha = \tau$ and $sRt'$.
   b) $\exists s \Rightarrow s_1 \xrightarrow{\alpha} s'$ such that $s_1Rt$ and $s'Rt$.

$R$ is called a **Rooted branching bisimulation**[a] if the following root condition is also satisfied:

- If $\text{root}(P) \xrightarrow{\alpha} s'$ for $\alpha \in \Sigma_\tau$, then there is a $t'$ with $\text{root}(Q) \xrightarrow{\alpha} t'$ and $s'Rt'$.
- If $\text{root}(Q) \xrightarrow{\alpha} t'$ for $\alpha \in \Sigma_\tau$, then there is an $s'$ with $\text{root}(P) \xrightarrow{\alpha} s'$ and $s'Rt'$.

[a] In this definition, the rules for Successful termination are omitted. The full rule set is defined in Definition 3.10

[b] Here, we use the symbol $\Rightarrow$ to mean a chain of $\tau$ actions, possibly none.

### 5.1.2 Lemmas and Theorems

We will now state some Lemmas to aid our proof of the validity of our translation.

**Theorem 5.1: Expansion of ACP merge**

The process

$$\partial_{H_0}(P \,||\, Q), \tag{5.2}$$

where $P, Q \in \text{ACP}_F^\tau$, $P$ is of the form $\alpha.P'$, $Q$ is of the form $\beta.Q'$[a], and $H_0$ is the set of working space operators as defined in Definition 4.1, can be expanded to the following process:

$$\begin{aligned}\partial_{H_0}(P \,||\, Q) =& \partial_{H_0}(\alpha).\partial_{H_0}(P' \,||\, Q)) \\ &+ \partial_{H_0}(\beta).\partial_{H_0}(Q' \,||\, P)) + \partial_{H_0}(\alpha \mid \beta).\partial_{H_0}(P' \,||\, Q')\end{aligned} \tag{5.3}$$

[a] Here, both $\alpha$ and $\beta$ are actions in the set $\Sigma \cup \{\tau\}$

*Proof.* We can derive the following equation from the axioms of ACP shown in B.1, starting by expanding the Axiom CM1.

$$
\begin{aligned}
\partial_{H_0}(P \,||\, Q) &= \partial_{H_0}\big(P \lfloor\!\lfloor Q + Q \lfloor\!\lfloor P + P \mid Q\big) \\
&= \partial_{H_0}\big(\alpha.P' \lfloor\!\lfloor Q + \beta.Q' \lfloor\!\lfloor P + \alpha.P' \mid \beta.Q'\big) \\
\text{CM3, CM7} &= \partial_{H_0}\big(\alpha.(P' \,||\, Q) + \beta.(Q' \,||\, P) + (\alpha \mid \beta).(P' \,||\, Q')\big) \\
\text{D3} &= \partial_{H_0}(\alpha.(P' \,||\, Q)) + \partial_{H_0}(\beta.(Q' \,||\, P)) + \partial_{H_0}\big((\alpha \mid \beta).(P' \,||\, Q')\big) \\
\text{D4} &= \partial_{H_0}(\alpha).\partial_{H_0}(P' \,||\, Q)) + \partial_{H_0}(\beta).\partial_{H_0}(Q' \,||\, P)) + \partial_{H_0}(\alpha \mid \beta).\partial_{H_0}(P' \,||\, Q')
\end{aligned}
$$

□

**Lemma 5.2**

For processes $P, Q \in \text{ACP}^{\tau}_{F}$, if $P$ is of the form $\alpha.P'$ where $\alpha \in \Sigma_0 \cup \{\tau\}$, and $Q$ is of the form $\beta.Q'$ where $\beta \in H_0$ (where $H_0$ is the set of working space operators as defined in Definition 4.1), and such that $\alpha \mid \beta$ is not defined on $|$, then

$$\partial_{H_0}(P \,||\, Q) = \alpha.\partial_{H_0}(P' \,||\, Q). \tag{5.4}$$

*Proof.* Starting from Theorem 5.1, we can derive the following equation:

$$
\begin{aligned}
\partial_{H_0}(P \,||\, Q) &= \partial_{H_0}(\alpha).\partial_{H_0}(P' \,||\, Q)) + \partial_{H_0}(\beta).\partial_{H_0}(Q' \,||\, P)) + \partial_{H_0}(\alpha \mid \beta).\partial_{H_0}(P' \,||\, Q') \\
&= \alpha.\partial_{H_0}(P' \,||\, Q) + \delta.\partial_{H_0}(Q' \,||\, P) + \partial_{H_0}(\delta).\partial_{H_0}(P' \,||\, Q') \\
&= \alpha.\partial_{H_0}(P' \,||\, Q) + \delta.\partial_{H_0}(Q' \,||\, P) + \delta.\partial_{H_0}(P' \,||\, Q') \\
&= \alpha.\partial_{H_0}(P' \,||\, Q)
\end{aligned}
$$

□

**Lemma 5.3**

For processes $P, Q \in \text{ACP}^{\tau}_{F}$, if $P$ is of the form $\alpha.P'$ and $Q$ is of the form $\beta.Q'$, where $\alpha, \beta \in \Sigma_0 \cup \{\tau\}$, and such that $\alpha \mid \beta$ is not defined on $|$, then

$$\partial_{H_0}(P \,||\, Q) = \alpha.\partial_{H_0}(P' \,||\, Q) + \beta.\partial_{H_0}(Q' \,||\, P). \tag{5.5}$$

*Proof.* Starting from Theorem 5.1, we can derive the following equation:

$$
\begin{aligned}
\partial_{H_0}(P \,||\, Q) &= \partial_{H_0}(\alpha).\partial_{H_0}(P' \,||\, Q) + \partial_{H_0}(\beta).\partial_{H_0}(Q' \,||\, P) + \partial_{H_0}(\alpha \mid \beta).\partial_{H_0}(P' \,||\, Q') \\
&= \alpha.\partial_{H_0}(P' \,||\, Q) + \beta.\partial_{H_0}(Q' \,||\, P) + \partial_{H_0}(\delta).\partial_{H_0}(P' \,||\, Q') \\
&= \alpha.\partial_{H_0}(P' \,||\, Q) + \beta.\partial_{H_0}(Q' \,||\, P) + \delta.\partial_{H_0}(P' \,||\, Q') \\
&= \alpha.\partial_{H_0}(P' \,||\, Q) + \beta.\partial_{H_0}(Q' \,||\, P)
\end{aligned}
$$

□

**Lemma 5.4**

For processes $P, Q \in \text{ACP}^{\tau}_{F}$, if $P$ is of the form $\alpha.P'$ and $Q$ is of the form $\beta.Q'$, where $\alpha, \beta \in H_0$ (where $H_0$ is the set of working space operators as defined in Definition 4.1), and such that $\alpha \mid \beta = \phi$, then

$$\partial_{H_0}(P \,||\, Q) = \phi.\partial_{H_0}(P' \,||\, Q'). \tag{5.6}$$

*Proof.* Starting from Theorem 5.1, we can derive the following equation:

$$\begin{aligned}
\partial_{H_0}(P \,||\, Q) &= \partial_{H_0}(\alpha).\partial_{H_0}(P' \,||\, Q) + \partial_{H_0}(\beta).\partial_{H_0}(Q' \,||\, P) + \partial_{H_0}(\alpha \mid \beta).\partial_{H_0}(P' \,||\, Q') \\
&= \delta.\partial_{H_0}(P' \,||\, Q) + \delta.\partial_{H_0}(Q' \,||\, P) + \partial_{H_0}(\phi).\partial_{H_0}(P' \,||\, Q') \\
&= \delta.\partial_{H_0}(P' \,||\, Q) + \delta.\partial_{H_0}(Q' \,||\, P) + \phi.\partial_{H_0}(P' \,||\, Q') \\
&= \phi.\partial_{H_0}(P' \,||\, Q')
\end{aligned}$$

□

**Lemma 5.5**

For processes $P, Q \in \text{ACP}^{\tau}_{F}$, if $P$ is of the form $\alpha.P$, and $Q$ is of the form $\tau^{*}.b.Q$ where $b \in H_0$ (where $H_0$ is the set of working space operators as defined in Definition 4.1), $\alpha$ does not communicate with $b$, and $\tau^{*}$ indicates a chain of consecutive $\tau$, possibly 0, then

$$\partial_{H_0}(a.P \,||\, \tau^{*}.b.Q) = \alpha.\partial_{H_0}(P) \,||\, \tau^{*}. \tag{5.7}$$

*Proof.* Starting from Lemma 5.3, we can derive the following equation:

$$\begin{aligned}
\partial_{H_0}(P \,||\, Q) &= \alpha.\partial_{H_0}((P' \,||\, Q)) + \beta.\partial_{H_0}((Q' \,||\, P)) \\
\partial_{H_0}(P \,||\, Q) &= \alpha.\partial_{H_0}((P' \,||\, Q)) + \tau.\partial_{H_0}((Q' \,||\, P))
\end{aligned}$$

Note that for $n$ number of consecutive $\tau$ moves in $Q$, $Q'$ has $n-1$ consecutive $\tau$ moves. We can recursively apply Lemma 5.3 on both sides of the processes, splitting them up into processes with progressively smaller and smaller number of $\tau$ moves, until we reach a state where we can apply Lemma 5.2, at which point we have the process

$$\begin{aligned}
\partial_{H_0}(P \,||\, Q) = {} & \underbrace{\tau.\tau.\cdots.\tau}_{n}.\big(\partial_{H_0}(\beta).\cdots + \alpha.\partial_{H_0}(P')\big) + \underbrace{\tau.\tau.\cdots.\tau}_{n-1}.\big(\partial_{H_0}(\beta).\cdots + \alpha.\partial_{H_0}(P')\big) \\
& + \cdots + \tau.\big(\partial_{H_0}(\beta).\cdots + \alpha.\partial_{H_0}(P')\big) + \big(\partial_{H_0}(\beta).\cdots + \alpha.\partial_{H_0}(P')\big)
\end{aligned}$$

Via Lemma 5.2, this can then be simplified to

$$\begin{aligned}
\partial_{H_0}(P \,||\, Q) = {} & \underbrace{\tau.\tau.\cdots.\tau}_{n}.\big(\alpha.\partial_{H_0}(P')\big) + \underbrace{\tau.\tau.\cdots.\tau}_{n-1}.\big(\alpha.\partial_{H_0}(P')\big) \\
& + \cdots + \tau.\big(\alpha.\partial_{H_0}(P')\big) + \big(\alpha.\partial_{H_0}(P')\big)
\end{aligned}$$

Since $\tau$ actions cannot communicate, this process is equivalent to

$$\partial_{H_0}(a.P \,||\, \tau^*.b.Q) = \alpha.\partial_{H_0}(P) \,||\, \tau^*.$$

□

**Lemma 5.6**

For a process $P \in \text{CSP}$ where $\mathscr{T}(P) \in \text{ACP}_F^\tau$ is strongly bisimilar to $P$, a process $\tau^* \,||\, \mathscr{T}(P)$ is Branching bisimilar to the process $P$. Here, I use the notation of $\tau^*$ indicating a chain of $\tau$, possibly 0.

*Proof.* We have the rule that

$$\tau \,||\, P =_{RBB} P$$

Now, simply take $P = \tau.P$ and inductively, this law will work for arbitrary $\tau$ actions. □

We can now work towards a proof that our translation is valid up to Rooted Branching bisimilarity.

## 5.2 Proof of Rooted Branching bisimilarity

We define a rooted branching bisimulation relation.

**Definition 5.7: Rooted branching bisimulation Relation**

Let $\mathbb{T}_{\text{CSP}}$ be the expressions in the language CSP, and $\mathbb{T}_{\text{ACP}_F^\tau}$ be expressions in the language $\text{ACP}_F^\tau$. We use the translation $\mathscr{T} : \mathbb{T}_{\text{CSP}} \to \mathbb{T}_{\text{ACP}_F^\tau}$ as defined in 4.13.

We now define a Rooted branching bisimulation between $\mathbb{T}_{\text{CSP}}$ and $\mathbb{T}_{\text{ACP}_F^\tau}$:

$$=_{RBB} := \{(P, \mathscr{T}(P)) \mid P \in \text{CSP}\}$$

### 5.2.1 Parallel Composition

The **Parallel Composition** operator $||_A$ is defined as the following equation:

$$\mathscr{T}(P \,||_A\, Q) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P)) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big)$$

Let $P, Q \in \text{CSP}$ be two processes. We want to show that $P \,||_A\, Q =_{RBB} \mathscr{T}(P \,||_A\, Q)$. i.e.: we want to show that any move in the original process can be replicable in the translated process up to rooted branching bisimulation. The different moves a process can take with the operator $||_A$ are defined in the SOS semantics shown in Table 3.2, and the relevant rules are also shown in S1. By exhausting all possible moves that $P \,||_A\, Q$ can take, and confirming that $\mathscr{T}(P \,||_A\, Q)$ can also take them, up to rooted branching bisimilarity, we will have shown that the equivalence holds true.

- Let $P'$ such that $P \xrightarrow{\alpha} P'$, and $\alpha \notin A$ for the target set $A$. $f_{\mathtt{syn}}$ will therefore not affect $\alpha$. In the domain of CSP, this results in the process:

  $$P \,||_A\, Q \xrightarrow{a} P' \,||_A\, Q$$

  We will show that the translation can also take this move. An action $\beta$ in $Q$ will be such that either $\beta \in H_0$, in which case we can apply Lemma 5.2,

  $$\begin{aligned}
  &\partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P)) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big) \\
  &\qquad = \alpha.\partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P')) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big) \\
  &\qquad \xrightarrow{\alpha} \partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P')) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big)
  \end{aligned}$$

  or $\beta$ will be in $\Sigma_0 \cup \tau$, in which case we can apply Lemma 5.3,

  $$\begin{aligned}
  &\partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P)) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big) \\
  &\qquad = \alpha.\partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P')) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big) \\
  &\qquad + \beta.\partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(Q')) \,||\, f_{\mathtt{syn}}(\mathscr{T}(P))\Big]\Big) \\
  &\qquad \xrightarrow{\alpha} \partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P')) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big)
  \end{aligned}$$

Note that the $\alpha$ and $\beta$ moves can move out of $f_{\mathtt{post}}$ because we specified that $a, b \in \Sigma_0 \cup \tau$. This move is therefore strongly bisimilar to $P' \,||_A\, Q$, even for internal actions.

- Let $Q'$ such that $Q \xrightarrow{\alpha} Q'$, and $\alpha \notin A$ for the target set $A$. This will give the same result as shown above, and is strongly bisimilar.

- Let $P'$ such that $P \xrightarrow{a} P'$ and $Q'$ s.t. $Q \xrightarrow{a} Q'$, where $a \in A$. Since $a_{\mathtt{syn}} \in H_0$ and $a_{\mathtt{syn}} \mid a_{\mathtt{syn}} = a_{\mathtt{post}}$, we apply Lemma 5.4:

$$\begin{aligned}\mathscr{T}(P \,||_A\, Q) &= \partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P)) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q))\Big]\Big) \\ &= \partial_{H_0}\Big(f_{\mathtt{post}}\Big[a_{\mathtt{syn}}.f_{\mathtt{syn}}(\mathscr{T}(P')) \,||\, a_{\mathtt{syn}}.f_{\mathtt{syn}}(\mathscr{T}(Q'))\Big]\Big) \\ &\xrightarrow{a} \partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{syn}}(\mathscr{T}(P')) \,||\, f_{\mathtt{syn}}(\mathscr{T}(Q'))\Big]\Big)\end{aligned}$$

  Note that the move $a$ results from $f_{\mathtt{post}}$ applied to $a_{\mathtt{post}}$, the result of the communication. This process is strongly bisimilar to $P' \,||_A\, Q'$.

Therefore, our translation for Parallel Composition is valid up to strong bisimilarity.

### 5.2.2 External choice

From Section 4.4.3, our translation of External Choice is:

$$\mathscr{T}(P \square Q) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[\Gamma[\mathscr{T}(P)] \,||\, \mathtt{choose} \,||\, \Gamma[\mathscr{T}(Q)]\Big]\Big) \tag{T9}$$

*Proof.* Let $P, Q \in$ CSP be two processes. We proceed similarly to the proof of Parallel Composition, by looking at the SOS rules laid out in 3.2, or in S2, and exhausting all possible moves that $P \square Q$ can take, confirming that $\mathscr{T}(P \square Q)$ can also take them up to rooted branching bisimilarity.

- Let $P'$ such that $P \xrightarrow{a} P'$. In the domain of CSP, this results in the process:

$$P \square Q \xrightarrow{a} P' \tag{5.8}$$

  Now working in ACP$^{\tau}_{F}$, we want to show the translation is valid up to rooted branching bisimulation, i.e. $\mathscr{T}(P') =_{RBB} \mathscr{T}(P')$, or in general, $\mathscr{T}(\mathcal{P}) =_{RBB} \mathcal{P}$ for any $\mathcal{P} \in$ CSP. From the definition of the Triggering operator O1, we can derive the following equation:

$$\Gamma(\mathscr{T}(P)) = a_{\mathtt{ini}}.\mathscr{T}(P') \tag{5.9}$$

  Since $\mathtt{choose} \in H_0$, and $a_{\mathtt{ini}} \mid \mathtt{choose} = a_{\mathtt{post}}$, we can derive the following process from Lemma 5.4:

$$\begin{aligned}\mathscr{T}(P \square Q) &= \partial_{H_0}\Big(f_{\mathtt{post}}\Big[\Gamma[\mathscr{T}(P)] \,||\, \mathtt{choose} \,||\, \Gamma[\mathscr{T}(Q)]\Big]\Big) \\ &= \partial_{H_0}\Big(f_{\mathtt{post}}\Big[a_{\mathtt{ini}}.\mathscr{T}(P') \,||\, \mathtt{choose} \,||\, \Gamma[\mathscr{T}(Q)]\Big]\Big) \\ &\xrightarrow{a} \partial_{H_0}\Big(f_{\mathtt{post}}\Big[\mathscr{T}(P') \,||\, \Gamma[\mathscr{T}(Q)]\Big]\Big)\end{aligned}$$

Note that the move $a$ results from $f_{\texttt{post}}$ applied to $a_{post}$, similarly to in the parallel composition operator. This is a common occurance, and from now on we will accept that this is the behaviour that will happen.

This is not yet a process that is comparable to $P'$, so we look at the next step. Due to the Triggering operator $\Gamma$ being applied to $Q$, the only communicatable action of a trace $Q_c$ of $Q$ will be one tagged with an `ini`, with some number of $\tau$ actions behind it. Via Lemma 5.5, any of the actions past $\tau^*$ will get restricted, leaving:

$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[\mathscr{T}(P') \,||\, \Gamma[\mathscr{T}(Q_c)]\Big]\Big) \implies \partial_{H_0}\Big(f_{\texttt{post}}\Big[\mathscr{T}(P') \,||\, \tau^*\Big]\Big) \implies \mathscr{T}(P') \,||\, \tau^*$$

for every trace $Q_c$ in $Q$. Via Lemma 5.6, this process is branching bisimilar to the process $\mathscr{T}(P')$. From this, we can see the union of every branch in $Q$ is at coarsest branching bisimilar, and therefore as the first action is related up to strong bisimilarity, taking an external action on $P$ is valid up to rooted branching bisimilarity.

- Let $P'$ such that $P \xrightarrow{\tau} P'$. In the domain of CSP, this results in the process:

$$P \,\square\, Q \xrightarrow{\tau} P' \,\square\, Q \tag{5.10}$$

  Now working in $\mathrm{ACP}^{\tau}_{F}$, we want to show that the translation is valid up to RBB, i.e. $\mathscr{T}(P' \,\square\, Q) =_{RBB} P' \,\square\, Q$. Via Lemma 5.2, we can now derive the following equation:

$$\begin{aligned}&\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma[\mathscr{T}(P)] \,||\, \texttt{choose} \,||\, \Gamma[\mathscr{T}(Q)]\Big]\Big) \xrightarrow{\tau}\\ &\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma[\mathscr{T}(P')] \,||\, \texttt{choose} \,||\, \Gamma[\mathscr{T}(Q)]\Big]\Big)\end{aligned}$$

  This process is strongly bisimilar to $P' \,\square\, Q$, therefore a $\tau$ action on $P$ is also valid up to rooted branching bisimilarity.

- The same logic from Item 1 and 2 can be applied in reverse to $Q$ and $P$ to also obtain processes that satisfy rooted branching bisimilarity.

We have now exhausted all cases, and therefore can conclude that our translation of CSP External Choice is valid up to rooted branching bisimilarity. □

### 5.2.3 Sliding Choice

From Section 4.4.4, our translation of the **Sliding Choice** operator $\triangleright$ is defined as the following equation:

$$\mathscr{T}(P \triangleright Q) = \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(P)) \,||\, \text{choose} \,||\, \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big) \tag{T10}$$

*Proof.* We proceed in the same manner as the previous operators.

- Let $P'$ such that $P \xrightarrow{a} P'$. Since $\texttt{choose} \in H_0$, and $\texttt{choose} \mid a_{\texttt{ini}} = a$, we apply Lemma 5.4 using Equation 5.9:

$$\begin{aligned}\mathscr{T}(P \triangleright Q) &= \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(P)) \,||\, \text{choose} \,||\, \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big)\\ &= \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[a_{\texttt{ini}}.\mathscr{T}(P') \,||\, \text{choose} \,||\, \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big)\\ &\xrightarrow{a} \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\mathscr{T}(P') \,||\, \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big)\end{aligned}$$

  Nothing in $\mathscr{T}(P')$ can communicate with the action $\texttt{shift}_{\texttt{ini}}$, therefore the process is strongly bisimilar to $P'$.

- Let $P'$ such that $P \xrightarrow{\tau} P'$. Since $\texttt{choose} \in H_0$, and $\tau$ actions cannot communicate, we apply Lemma 5.2:

$$\begin{aligned}\mathscr{T}(P \triangleright Q) &= \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(P)) \,||\, \text{choose} \,||\, \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big)\\ &= \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\tau.\Gamma(\mathscr{T}(P')) \,||\, \text{choose} \,||\, \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big)\\ &\xrightarrow{\tau} \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(P')) \,||\, \texttt{choose} \,||\, \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big)\end{aligned}$$

  This process is strongly bisimilar to $P' \triangleright Q$.

- Let $Q'$ such that $Q \xrightarrow{\alpha} Q'$. Since $\texttt{choose} \in H_0$, and the first action of the process must be $\texttt{shift}_{\texttt{ini}}$, we apply Lemma 5.4:

$$\begin{aligned}\mathscr{T}(P \triangleright Q) &= \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(P)) \,||\, \text{choose} \,||\, \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big)\\ &\xrightarrow{\tau} \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(P')) \,||\, \texttt{shift}.\mathscr{T}(Q))\Big]\Big)\Big)\end{aligned}$$

  This process is branching bisimilar to $Q$. The action $\texttt{shift}$ will be abstracted by the $\tau$, which is a strongly bisimilar action, but the process $\Gamma(\mathscr{T}(P))$ can still have $\tau$ actions, which via Lemma 5.5 will cause the process to derive to

$$\tau^* \,||\, \tau.\mathscr{T}(Q). \tag{5.11}$$

  which is rooted branching bisimilar via Lemma 5.6.

Therefore, our translation for Sliding Choice is valid up to rooted branching bisimulation.

□

### 5.2.4 Interrupt

From Section 4.4.5, our translation of the **Interrupt** operator $\triangle$ is defined as the following equation:

$$\mathscr{T}(P \triangle Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P)) \,||\, \Pi \,||\, \Gamma(\mathscr{T}(Q))\Big]\Big) \tag{T11}$$

*Proof.* We proceed in the same manner as the previous operators.

- Let $P'$ such that $P \xrightarrow{a} P'$. Since every visible action in $P$ is tagged $a_{\texttt{origin}}$, which can always communicate with $\texttt{origin} \in H_0$, we can always apply Lemma 5.4 (Note that $\Pi \xrightarrow{\texttt{origin}} \Pi$)

$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P))\,||\,\Pi\,||\,\Gamma(\mathscr{T}(Q))\Big]\Big) \xrightarrow{a}$$
$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P'))\,||\,\Pi\,||\,\Gamma(\mathscr{T}(Q))\Big]\Big)$$

  which is Strongly bisimilar to $P' \triangle Q$.

- Let $P'$ such that $P \xrightarrow{\tau} P'$. Since $\tau$ actions can never communicate, and we have that $\texttt{split}, \texttt{origin} \in H_0$, we can always apply Lemma 5.2:

$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P))\,||\,\Pi\,||\,\Gamma(\mathscr{T}(Q))\Big]\Big) \xrightarrow{\tau}$$
$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P'))\,||\,\Pi\,||\,\Gamma(\mathscr{T}(Q))\Big]\Big)$$

  which is Strongly bisimilar to $P' \triangle Q$.

- Let $Q'$ such that $Q \xrightarrow{a} Q'$. At any step in $\Pi$, there is the possibility to take a $\texttt{split}$ step, and communicate with $a_{\texttt{ini}}$. Therefore, we can then apply Lemma 5.4:

$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P))\,||\,\Pi\,||\,\Gamma(\mathscr{T}(Q))\Big]\Big) \xrightarrow{a}$$
$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P))\,||\,\mathscr{T}(Q')\Big]\Big)$$

  Via Lemma 5.5, This process reduces to

$$\tau^*\,||\,\mathscr{T}(Q')$$

  which is Branching bisimilar to $P \triangle Q'$ via Lemma 5.6.

- Let $Q'$ such that $Q \xrightarrow{\tau} Q'$. Since $\tau$ actions can never communicate, and we have that $\texttt{split}, \texttt{origin} \in H_0$, we can always apply Lemma 5.2:

$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P))\,||\,\Pi\,||\,\Gamma(\mathscr{T}(Q))\Big]\Big) \xrightarrow{\tau}$$
$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P))\,||\,\Pi\,||\,\Gamma(\mathscr{T}(Q'))\Big]\Big)$$

  This process is Strongly bisimilar to $P \triangle Q'$.

We have now exhausted all cases, and therefore we can conclude that our translation of CSP Interrupt is valid up to rooted branching bisimulation. □

### 5.2.5 Throw

From Section 4.4.6, the **Throw** operator $\Theta_A$ is defined as the following equation:

$$\mathscr{T}(P\,\Theta_A\,Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P))||\Pi.\mathscr{T}(Q)\Big]\Big). \tag{T12}$$

*Proof.* We proceed in the same manner as the previous operators.

- Let $P'$ such that $P \xrightarrow{\tau} P'$. Therefore, we have:
$$f_{\texttt{split}}(\mathscr{T}(P)) = \tau.f_{\texttt{split}}(\mathscr{T}(P'))$$
Since $\texttt{split}, \texttt{origin} \in H_0$, and $\tau$ cannot communicate, we apply Lemma 5.2:
$$\begin{aligned}\mathscr{T}(P\,\Theta_A\,Q) =&\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P))||\Pi.\mathscr{T}(Q)\Big]\Big)\\ =&\partial_{H_0}\Big(f_{\texttt{post}}\Big[\tau.f_{\texttt{split}}(\mathscr{T}(P'))||\Pi.\mathscr{T}(Q)\Big]\Big) \xrightarrow{\tau}\\ &\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P'))||\Pi.\mathscr{T}(Q)\Big]\Big)\end{aligned}$$
This process is strongly bisimilar to $P'\,\Theta_A\,Q$.
- Let $P'$ such that $P \xrightarrow{a} P'$, and $a \notin A$ for the target set $A$. We will have that:
$$f_{\texttt{split}}(\mathscr{T}(P)) = a_{\texttt{origin}}.f_{\texttt{split}}(\mathscr{T}(P'))$$
Since $\texttt{split}, \texttt{origin} \in H_0$, and $a_{\texttt{origin}} \mid \texttt{origin} = a$, we apply Lemma 5.4:
$$\begin{aligned}\mathscr{T}(P\,\Theta_A\,Q) =&\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P))||\Pi.\mathscr{T}(Q)\Big]\Big)\\ =&\partial_{H_0}\Big(f_{\texttt{post}}\Big[a_{\texttt{origin}}.f_{\texttt{split}}(\mathscr{T}(P'))||\Pi.\mathscr{T}(Q)\Big]\Big) \xrightarrow{a}\\ &\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P'))||\Pi.\mathscr{T}(Q)\Big]\Big)\end{aligned}$$
This process is strongly bisimilar to $P'\,\Theta_A\,Q$.
- Let $P'$ such that $P \xrightarrow{a} P'$, and $a \in A$ for the target set $A$. We will have that:
$$f_{\texttt{split}}(\mathscr{T}(P)) = a_{\texttt{split}}.f_{\texttt{split}}(\mathscr{T}(P'))$$
Since $\texttt{split}, \texttt{origin} \in H_0$, and $a_{\texttt{split}} \mid \texttt{split} = a$, we apply Lemma 5.4:
$$\begin{aligned}\mathscr{T}(P\,\Theta_A\,Q) =&\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P))||\Pi.\mathscr{T}(Q)\Big]\Big)\\ =&\partial_{H_0}\Big(f_{\texttt{post}}\Big[a_{\texttt{split}}.f_{\texttt{split}}(\mathscr{T}(P'))||\Pi.\mathscr{T}(Q)\Big]\Big) \xrightarrow{a}\\ &\partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P'))||\mathscr{T}(Q)\Big]\Big)\end{aligned}$$
Via Lemma 5.5, this reduces down to the equation:
$$\tau^*||\mathscr{T}(Q)$$
This is rooted branching bisimilar to $Q$ via Lemma 5.6.

We have now exhausted all cases, and therefore we can conclude that our translation of CSP Throw is valid up to rooted branching bisimulation.

□

### 5.2.6 Generalising

From these proofs, we have shown that every translation of the CSP operators are valid up to rooted branching bisimilarity.

# Chapter 6

# Conclusion

## 6.1 Results

In this paper, we have demonstrated a translation from the language CSP to the language $ACP_F^\tau$, an extension of the language $ACP_\tau$ to include a Functional Renaming operator. We have shown that this translation is valid up to rooted branching bisimulation, which is Compositional in the language ACP and its related extensions. We can therefore say that $ACP_F^\tau$ is **at least as expressive as** CSP **up to rooted branching bisimulation**.

## 6.2 Limitations and Future Work

In our encoding, we are limited by the fact that internal actions will remain in communications, and hence hindering the possibility of a Strong Bisimulation. The behaviour of abstraction being a one-way operator, and $\tau$ actions being forgetful is an intentional and inherent part of $ACP_\tau$ from an axiomatic level. This leads me to believe that rooted branching bisimilarity is the finest relation there is between variants of CSP and ACP.

Further work could be done to study the relationship between these two languages, as there are many fragments and extensions to the language ACP and there may be a different approach which avoids this conundrum, with a valid solution up to strong bisimilarity possibly existing for some fragment of ACP.

# Bibliography


[1] ABADI, Martín ; GORDON, Andrew D.: A Calculus for Cryptographic Protocols: The Spi Calculus. In: *Information and Computation* 148 (1999), Januar, Nr. 1, S. 1–70. – ISSN 0890-5401

[2] AUSTRY, Didier ; BOUDOL, Gérard: Algèbre de processus et synchronisation. In: *Theoretical Computer Science* 30 (1984), Nr. 1, S. 91–131. – ISSN 03043975

[3] BAETEN, J. C. M.: A Brief History of Process Algebra. In: *Theoretical Computer Science* 335 (2005), Mai, Nr. 2, S. 131–146. – ISSN 0304-3975

[4] BAETEN, J. C. M. ; WEIJLAND, W. P.: *Process Algebra*. Cambridge : Cambridge University Press, 1990 (Cambridge Tracts in Theoretical Computer Science)

[5] BERGSTRA, J. A. ; KLOP, J. W.: Process Algebra for Synchronous Communication. In: *Information and Control* 60 (1984), Januar, Nr. 1, S. 109–137. – ISSN 0019-9958

[6] BERGSTRA, J. A. ; KLOP, J. W.: ACP$\tau$ a Universal Axiom System for Process Specification. In: WIRSING, Martin (Hrsg.) ; BERGSTRA, Jan A. (Hrsg.): *Algebraic Methods: Theory, Tools and Applications*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1989, S. 445–463. – ISBN 978-3-540-46758-8

[7] BROOKES, S. D. ; HOARE, C. A. R. ; ROSCOE, A. W.: A Theory of Communicating Sequential Processes. In: *J. ACM* 31 (1984), Juni, Nr. 3, S. 560–599. – ISSN 0004-5411

[8] CARDELLI, Luca ; GORDON, Andrew D.: Mobile Ambients. In: NIVAT, Maurice (Hrsg.): *Foundations of Software Science and Computation Structures*. Berlin, Heidelberg : Springer, 1998, S. 140–155. – ISBN 978-3-540-69720-6

[9] DE SIMONE, Robert: Higher-Level Synchronising Devices in Meije-SCCS. In: *Theoretical Computer Science* 37 (1985), S. 245–267. – ISSN 03043975

[10] FOKKINK, Wan: Rooted Branching Bisimulation as a Congruence. In: *Journal of Computer and System Sciences* 60 (2000), Februar, Nr. 1, S. 13–37. – ISSN 0022-0000

[11] GARAVEL, Hubert: Revisiting Sequential Composition in Process Calculi. In: *Journal of Logical and Algebraic Methods in Programming* 84 (2015), Nr. 6, S. 742–762. – ISSN 2352-2208

[12] GORLA, Daniele: Towards a Unified Approach to Encodability and Separation Results for Process Calculi. In: *Information and Computation* 208 (2010), September, Nr. 9, S. 1031–1053. – ISSN 0890-5401

[13] MILNER, Robin: *Lecture Notes in Computer Science*. Bd. 92: *A Calculus of Communicating Systems*. Berlin, Heidelberg : Springer, 1980. – ISBN 978-3-540-10235-9 978-3-540-38311-6

[14] MILNER, Robin ; PARROW, Joachim ; WALKER, David: A Calculus of Mobile Processes, I. In: *Information and Computation* 100 (1992), Nr. 1, S. 1–40. – ISSN 0890-5401

[15] OLDEROG, E. R. ; HOARE, C. A. R.: Specification-Oriented Semantics for Communicating Processes. In: *Acta Informatica* 23 (1986), April, Nr. 1, S. 9–66. – ISSN 0001-5903, 1432-0525

[16] PARROW, J. ; VICTOR, B.: The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. In: *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.98CB36226)*, Juni 1998, S. 176–185. – ISSN 1043-6871

[17] PARROW, Joachim: Expressiveness of Process Algebras. In: *Electronic Notes in Theoretical Computer Science* 209 (2008), April, S. 173–186. – ISSN 1571-0661

[18] PETERS, Kirstin: Comparing Process Calculi Using Encodings. In: *Electronic Proceedings in Theoretical Computer Science* 300 (2019), August, S. 19–38. – ISSN 2075-2180

[19] PLOTKIN, Gordon: A Structural Approach to Operational Semantics. In: *The Journal of Logic and Algebraic Programming* 60–61 (2004), Juli, S. 17–139. – ISSN 15678326

[20] ROSCOE, A.W.: *Understanding Concurrent Systems*. London : Springer, 2010 (Texts in Computer Science). – ISBN 978-1-84882-257-3 978-1-84882-258-0

[21] VAN GLABBEEK, Rob: On the Expressiveness of ACP. In: PONSE, A. (Hrsg.) ; VERHOEF, C. (Hrsg.) ; VAN VLIJMEN, S. F. M. (Hrsg.): *Algebra of Communicating Processes*. London : Springer London, 1995, S. 188–217. – ISBN 978-1-4471-2120-6

[22] VAN GLABBEEK, Rob: A Theory of Encodings and Expressiveness (Extended Abstract). In: BAIER, Christel (Hrsg.) ; LAGO, Ugo D. (Hrsg.): *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings* Bd. 10803, Springer, 2018, S. 183–202

[23] VAN GLABBEEK, Rob: A Theory of Encodings and Expressiveness. (2024)

[24] VRANCKEN, J.L.M.: The Algebra of Communicating Processes with Empty Process. In: *Theoretical Computer Science* 177 (1997), Mai, Nr. 2, S. 287–328. – ISSN 03043975

# Appendix A

# Diagrams

## A.1 Examples of Translations

In this section I will provide visual aids for the encoding of translations for the different operators of the language CSP. An overview of the translation is shown below:

**Recall 1.1: Translation**

From Definition 4.13:

$$
\begin{aligned}
\mathscr{T}(STOP) &= \delta \\
\mathscr{T}(\mathrm{div}) &= \langle X \mid X = \tau.X \rangle \\
\mathscr{T}(a \to P) &= a.\mathscr{T}(P) \\
\mathscr{T}(P \sqcap Q) &= \tau.\mathscr{T}(P) + \tau.\mathscr{T}(Q) \\
\mathscr{T}(P \Box Q) &= \partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma[\mathscr{T}(P)] \mid\mid \texttt{choose} \mid\mid \Gamma[\mathscr{T}(Q)]\Big]\Big) \\
\mathscr{T}(P \triangleright Q) &= \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(P)) \mid\mid \mathrm{choose} \mid\mid \texttt{shift}_{\texttt{ini}}.\mathscr{T}(Q))\Big]\Big)\Big) \\
\mathscr{T}(P \mid\mid_A Q) &= \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{syn}}(\mathscr{T}(P)) \mid\mid f_{\texttt{syn}}(\mathscr{T}(Q))\Big]\Big) \\
\mathscr{T}(P \backslash A) &= \partial_A \mathscr{T}(P) \\
\mathscr{T}(f(P)) &= f(\mathscr{T}(P)) \\
\mathscr{T}(P \triangle Q) &= \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\mathscr{T}(P)) \mid\mid \Pi \mid\mid \Gamma(\mathscr{T}(Q))\Big]\Big) \\
\mathscr{T}(P \Theta_A Q) &= \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{split}}(\mathscr{T}(P)) \mid\mid \Pi.\mathscr{T}(Q)\Big]\Big)
\end{aligned}
$$

### A.1.1 External Choice

Below is a translation of the process:

$$\mathscr{T}(a.a\Box b) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[\Gamma[\mathscr{T}(a.a)] \mid\mid \mathtt{choose} \mid\mid \Gamma[\mathscr{T}(b)]\Big]\Big)$$

The result is strongly bisimilar to its original CSP process, with the translated process being $a.a+b$.

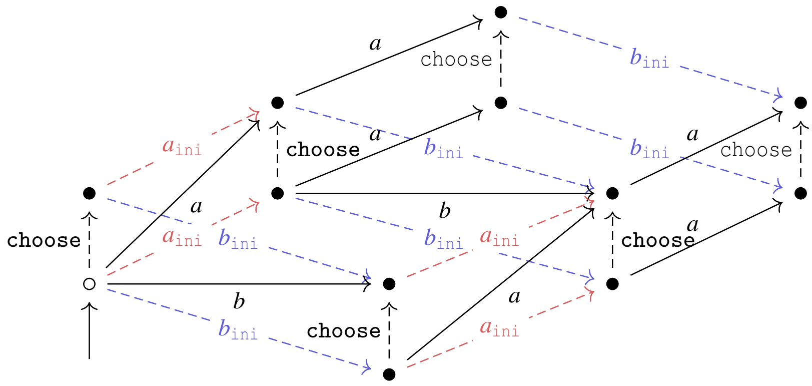



Figure A.1: Translation of $a.a\Box b$

Below is a translation of the process:

$$\mathscr{T}(\tau.a\Box b) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[\Gamma[\mathscr{T}(\tau.a)] \mid\mid \mathtt{choose} \mid\mid \Gamma[\mathscr{T}(b)]\Big]\Big)$$

The result is rooted branching bisimilar to its original CSP process, with the translated process being $b.\tau+\tau.(a+b)$ instead of $b+\tau.(a+b)$.

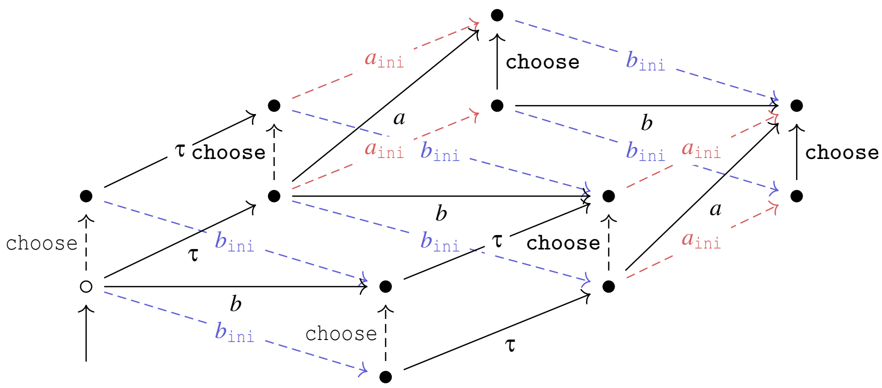



Figure A.2: Translation of $\alpha.a\Box b$

### A.1.2 Sliding Choice

Below is a translation of the process:

$$\mathscr{T}(b \triangleright a) = \tau_{\{\mathtt{shift}\}}\Big(\partial_{H_0}\Big(f_{\mathtt{post}}\Big[\Gamma(\mathscr{T}(b)) \mid\mid \mathrm{choose} \mid\mid \mathtt{shift}_{\mathtt{ini}}.\mathscr{T}(a))\Big]\Big)\Big)$$

The result is strongly bisimilar to its original CSP process, with the translated process being $b + \tau.a$.

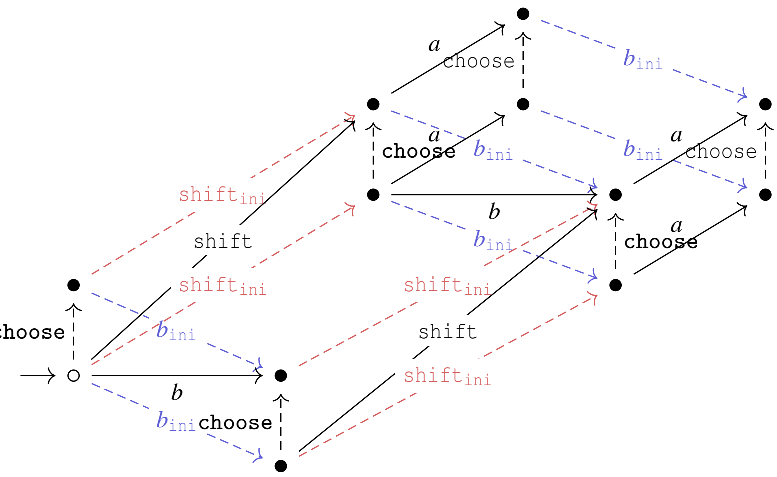



Figure A.3: Translation of $b \triangleright a$

Below is a translation of the process:

$$\mathscr{T}(\tau.b \triangleright a) = \tau_{\{\texttt{shift}\}}\Big(\partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma(\mathscr{T}(\tau.b))\,||\,\text{choose}\,||\,\texttt{shift}_{\texttt{ini}}.\mathscr{T}(a))\Big]\Big)\Big)$$

The result is Rooted Branching Bisimilar to its original CSP process, with the translated process being $\tau.(b+\tau.a)+\tau.(a.\tau+\tau.a)$ instead of $\tau.(b+\tau.a)+\tau.a$

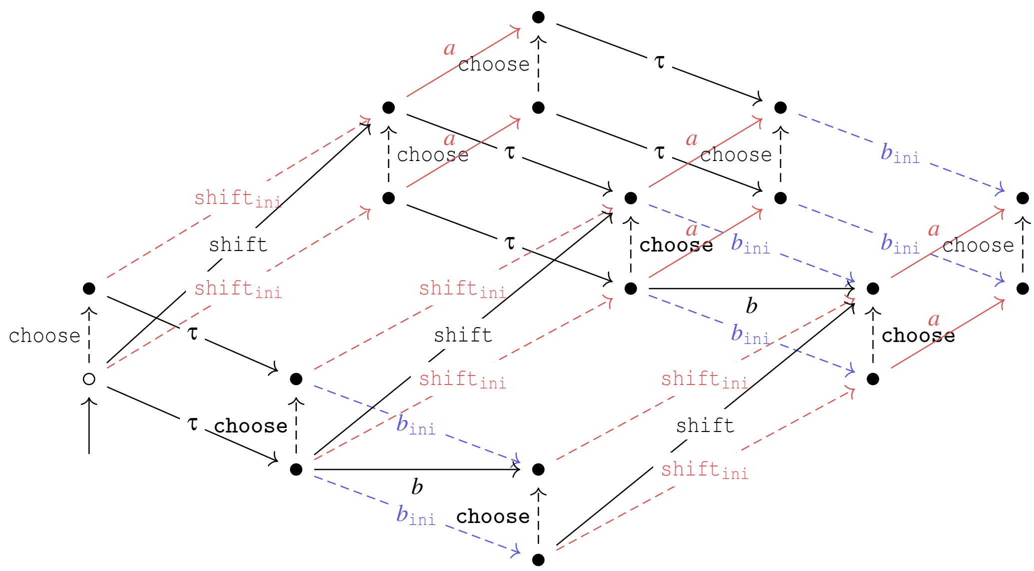



Figure A.4: Translation of $\tau.b \triangleright a$

### A.1.3 Interrupt

Below is a translation of the process:

$$\mathscr{T}(a \triangle b) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[\big(f_{\texttt{origin}}(a)\,||\,\Pi\big)\,||\,\Gamma(b)\Big]\Big)$$

The result is strongly bisimilar to its original CSP process, with the translated process being $a.b+b$. Here, the action `origin` is shortened to $O$, and the action `split` is shortened to $S$.

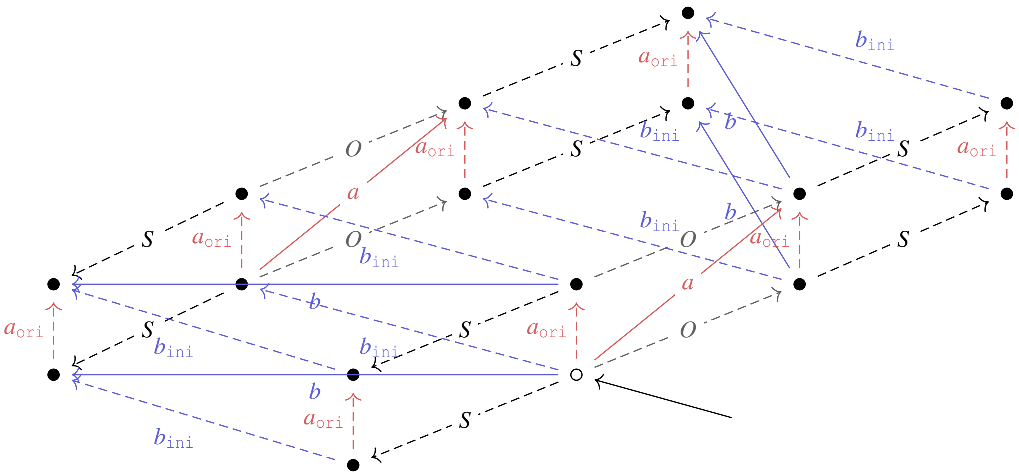



Figure A.5: Translation of $\mathscr{T}(a \triangle b)$

Below is a simple counterexample to the Interrupt operator on the process:

$$\mathscr{T}(\tau \triangle b) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{origin}}(\tau)\,||\,\Pi\,||\,\Gamma(b)\Big]\Big)$$

The result is rooted branching bisimilar to its original CSP process, with the translated process being $\tau.b+b.\tau$ instead of $b+\tau.b$

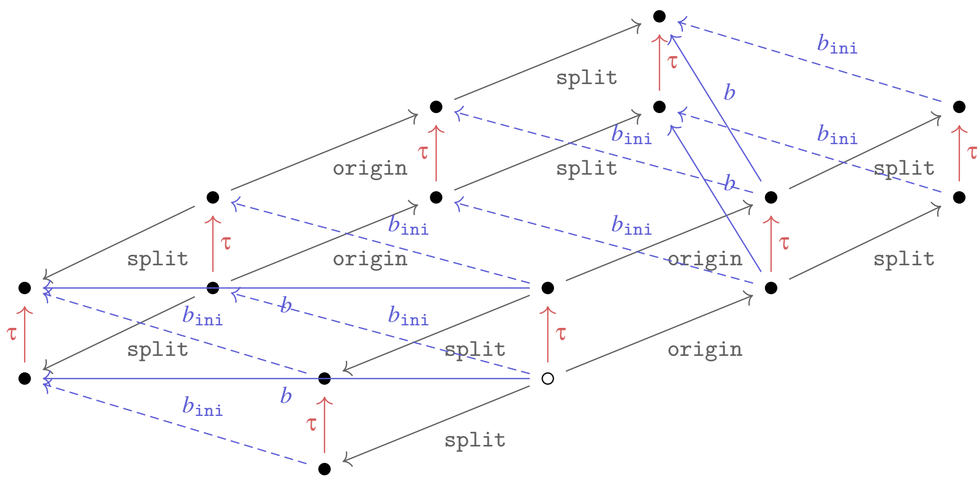



Figure A.6: Translation of $\tau \triangle b$

This should yield process graph $a$ in Figure A.7, however it yields process graph $b$ instead.:


(a) The process after Restriction

(b) The final reformatted process

Figure A.7: Intended vs actual result of the Interrupt operator on translation of $\tau \triangle b$

Below is a translation for the process:

$$\mathscr{T}(\tau.a\triangle\tau.b) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[\big(f_{\texttt{origin}}(\tau.a)\,||\,\Pi\big)\,||\,\Gamma(\tau.b)\Big]\Big)$$

The result is rooted branching bisimilar to its original CSP process., with the process being $\tau.(a.\tau.b+\tau(a.b+b)+\tau.(b.\tau+\tau.(b+a.b))$ instead of $\tau.a.\tau.b+\tau.\tau.b+\tau.b$. Here, the action `origin` is shortened to $O$, and the action `split` is shortened to $S$. A more comprehensive view is shown below.

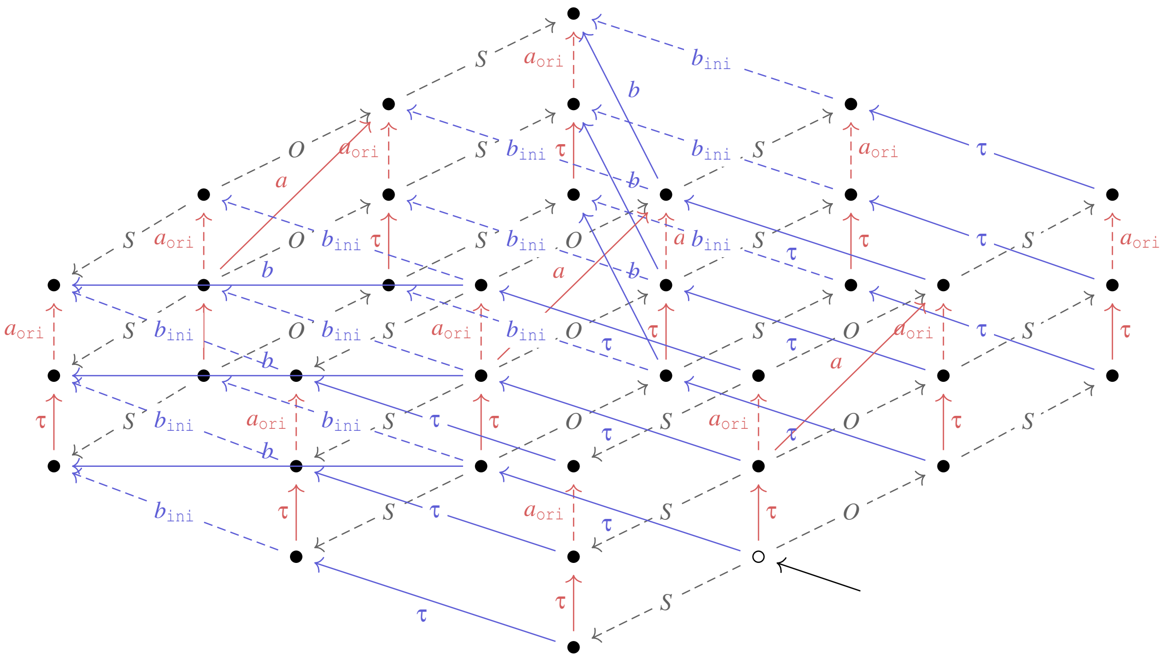
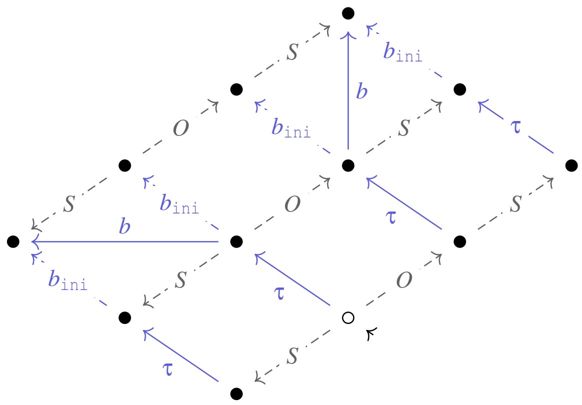
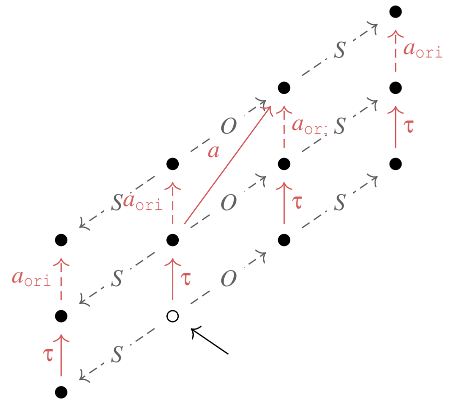


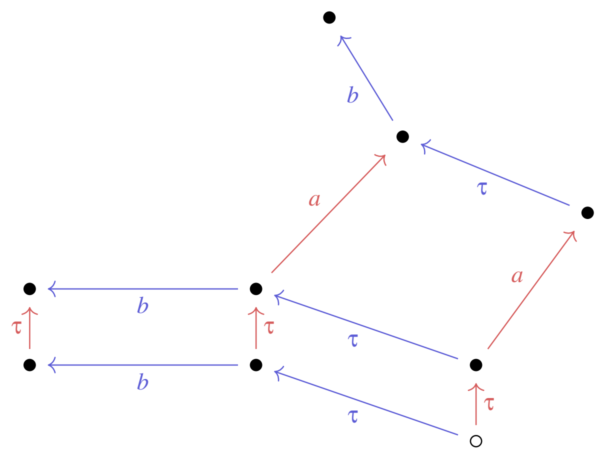

Figure A.8: All-in-one view of the translation of $\tau.a\triangle\tau.b$



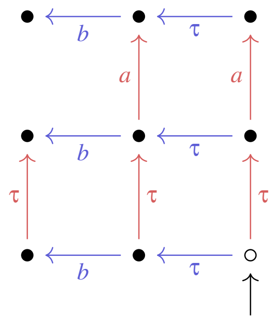

(a) The process $\Pi$ merged with the process $\Gamma(\tau.b)$




(b) The process $\Pi$ merged with the process $f_{\mathtt{origin}}(\tau.a)$




(c) The process after Restriction




(d) The final reformatted process

Figure A.9: In-depth view of the translation of $\mathscr{T}(\tau.a\triangle\tau.b)$

### A.1.4 Throw

Below is a translation of the process

$$\mathscr{T}(a.a.b.a\Theta_A q_1) = \partial_{H_0}\Big(f_{\mathtt{post}}\Big[f_{\mathtt{split}}(\mathscr{T}(a.a.b.a))||\Pi.\mathscr{T}(q_1)\Big]\Big)$$

Where the target set $A$ is $\{b\}$. The result is strongly bisimilar to its original CSP process.

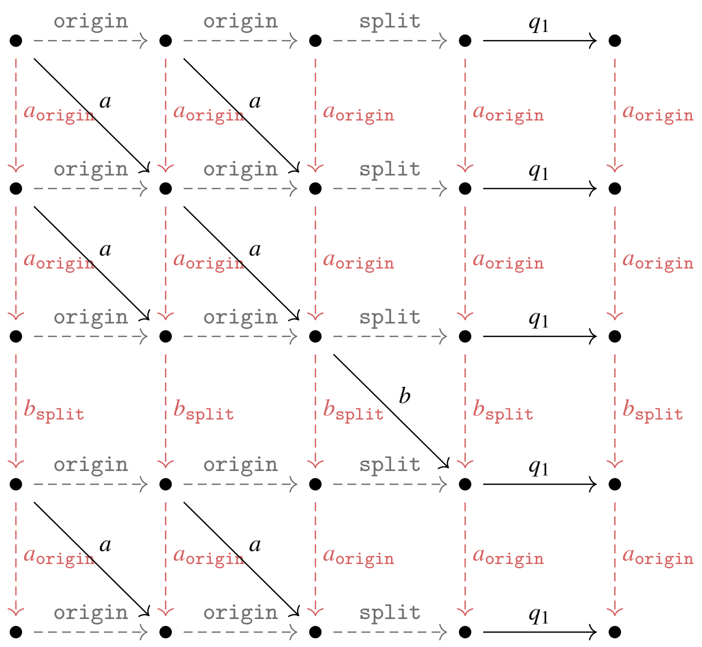
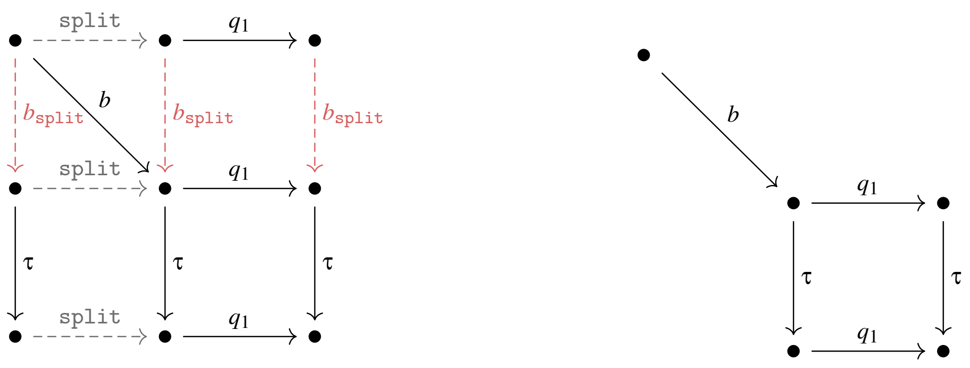



Figure A.10: Translation of $a.a.b.a\,\Theta_A\,q_1$

Below is a counterexample translation of the process

$$\mathscr{T}(a.a.b.a\,\Theta_A\,q_1) = \partial_{H_0}\left(f_{\texttt{post}}\left[f_{\texttt{split}}(\mathscr{T}(a.a.b.a))||\Pi.\mathscr{T}(q_1)\right]\right)$$

Where the target set $A$ is $\{b\}$. The result is rooted branching bisimilar to its original CSP process, with the process being translated to $b.(\tau + q_1)$ instead of $b.q_1$




(a) The process after Restriction

(b) The translated result

Figure A.11: Intended vs actual result of the process $\mathscr{T}(\tau \triangle b)$

### A.1.5 Parallel Composition

Below is a translation of the process:

$$\mathscr{T}(\tau.a.b \,||_A\, a.c.c) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[f_{\texttt{syn}}(\mathscr{T}(\tau.a.b))\,||\,f_{\texttt{syn}}(\mathscr{T}(a.c.c))\Big]\Big)$$

Where the target set $A$ is $\{a\}$. The result is Strongly Bisimilar to its original CSP process, with the translated process being

$$\tau.a.(b.c.c + c.b.c + c.c.b)$$

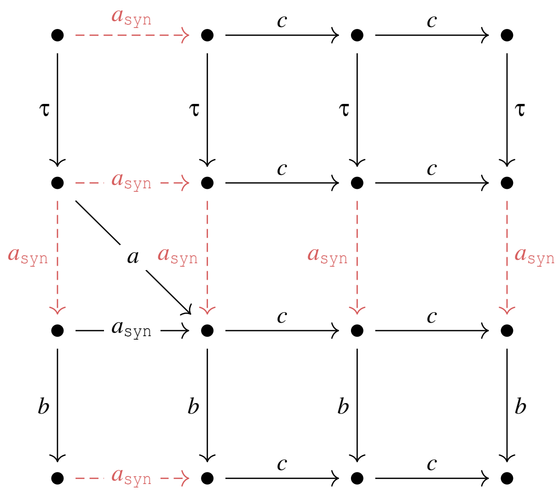



Figure A.12: Translation of $\tau.a.b \,||_A\, a.c.c$

# Appendix B

# Extended Definitions

## B.1 Extended Definitions

### B.1.1 Additional Axioms for the langauge $\mathrm{ACP}_\tau$

Below is a subset of the Axioms we use in Section 5.2 taken from [6]

| **Axioms of** $\mathrm{ACP}_\tau$ |
|---|
| **CM1**: $P \,\|\, Q = P \llcorner\!\!\lfloor Q + Q \llcorner\!\!\lfloor P + P \mid Q$ |
| **CM3**: $a.P \llcorner\!\!\lfloor Q = a.(P \llcorner\!\!\lfloor Q)$ |
| **CM7**: $a.P \mid b.Q = (a \mid b)(P \,\|\, Q)$ |
| **C3**: $\delta \mid a = \delta$ |
| **D3**: $\partial_H(P+Q) = \partial_H(P) + \partial_H(Q)$ |

Table B.1: Small Subset of Axioms of $\mathrm{ACP}_\tau$ Merge

### B.1.2 Operational Semantics for $\mathrm{ACP}_F^\tau$

Shown in Figure B.2 is the full set of Operational Semantics for the language $\mathrm{ACP}_F^\tau$.

$$a \xrightarrow{\alpha} \checkmark \qquad \frac{P \xrightarrow{\alpha} \checkmark}{P+Q \xrightarrow{\alpha} \checkmark} \qquad \frac{Q \xrightarrow{\alpha} \checkmark}{P+Q \xrightarrow{\alpha} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'} \qquad \frac{P \xrightarrow{\alpha} \checkmark}{P \cdot Q \xrightarrow{\alpha} Q} \qquad \frac{P \xrightarrow{\alpha} P'}{P \cdot Q \xrightarrow{\alpha} P' \cdot Q} \qquad \frac{P \xrightarrow{\alpha} \checkmark}{P \,||\, Q \xrightarrow{\alpha} Q}$$

$$\frac{Q \xrightarrow{\alpha} \checkmark}{P \,||\, Q \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P \,||\, Q \xrightarrow{\alpha} P' \,||\, Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \,||\, Q \xrightarrow{\alpha} P \,||\, Q'}$$

$$\frac{P \xrightarrow{\alpha} \checkmark \quad Q \xrightarrow{b} \checkmark \quad a|b=c}{P \,||\, Q \xrightarrow{c} \checkmark} \qquad \frac{P \xrightarrow{\alpha} \checkmark \quad Q \xrightarrow{b} Q' \quad a|b=c}{P \,||\, Q \xrightarrow{c} Q'}$$

$$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{b} \checkmark \quad a|b=c}{P \,||\, Q \xrightarrow{c} P'} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{b} Q' \quad a|b=c}{P \,||\, Q \xrightarrow{c} P' \,||\, Q'}$$

$$\frac{P \xrightarrow{\alpha} \checkmark}{P \mathbin{\lfloor\!\lfloor} Q \xrightarrow{\alpha} Q} \qquad \frac{P \xrightarrow{\alpha} P'}{P \mathbin{\lfloor\!\lfloor} Q \xrightarrow{\alpha} P' \,||\, Q} \qquad \frac{P \xrightarrow{\alpha} \checkmark \quad Q \xrightarrow{b} \checkmark \quad a|b=c}{P \mid Q \xrightarrow{c} \checkmark}$$

$$\frac{P \xrightarrow{\alpha} \checkmark \quad Q \xrightarrow{b} Q' \quad a|b=c}{P \mid Q \xrightarrow{c} Q'} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{b} \checkmark \quad a|b=c}{P \mid Q \xrightarrow{c} P'}$$

$$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{b} Q' \quad a|b=c}{P \mid Q \xrightarrow{c} P' \,||\, Q'} \qquad \frac{P \xrightarrow{\alpha} \checkmark \quad (\alpha \in I)}{\tau_I(P) \xrightarrow{\tau} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P' \quad (\alpha \in I)}{\tau_I(P) \xrightarrow{\tau} \tau_I(P')}$$

$$\frac{P \xrightarrow{\alpha} \checkmark \quad (\alpha \in I)}{\tau_I(P) \xrightarrow{\alpha} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin I)}{\tau_I(P) \xrightarrow{\alpha} \tau_I(P')} \qquad \frac{P \xrightarrow{\alpha} \checkmark \quad (\alpha \notin H)}{\partial_H(P) \xrightarrow{\alpha} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin H)}{\partial_H(P) \xrightarrow{\alpha} \partial_H(P')}$$

$$\frac{\langle \mathcal{S}_X \mid \mathcal{S} \rangle \xrightarrow{\alpha} \checkmark}{\langle X \mid \mathcal{S} \rangle \xrightarrow{\alpha} \checkmark} \qquad \frac{\langle \mathcal{S}_X \mid \mathcal{S} \rangle \xrightarrow{\alpha} P'}{\langle X \mid \mathcal{S} \rangle \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} \checkmark}{f(P) \xrightarrow{f(a)} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P'}{f(P) \xrightarrow{f(a)} f(P')}$$

Table B.2: Extended Structural operational semantics of the language $\mathrm{ACP}^{\tau}_{F}$. Compared to Table 3.1, this includes additional rules for Successful Termination, $\checkmark$, which are highlighted in red.