A translation from CSP to ACP

Leon Lee

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

2025

Abstract

A popular technique for representing Concurrent Systems is categorised into models called Process Algebras. Many Process Algebras exist, and efforts have been made to contextualise different Algebra to 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, that is valid up to a Rooted Branching Bisimulation equivalence.

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

Any acknowledgements go here.

Table of Contents

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 1930s with Turing Machines, and the λ-calculus. Systems in real life are rarely sequential however, and usually involve multiple processes acting simultaneously, sometimes even synchronising 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 multi-dimensional 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 goal of Expressiveness we are trying to model is to create a hierarchy of different Process Algebra 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 heirarchy 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 demonstraction of a simple Process Algebra, as well as 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", and the method we will be taking to reach it. In **Chapter 4**, I will define the actual Translation between CSP and ACP, and any relevant assistance that will be needed to 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 results and conclusion to the paper.

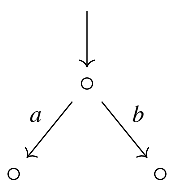
Chapter 2

Background

2.1 Process Algebra

Concurrency has been studied in many different ways, though with the earliest the 1960s with 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 concurrent systems. (Baeten, 2005) 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.

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". Process algebras can typically be modelled in 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 way:



Where 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. This could be thought of as the actions "Turn Right", leading to a shop, and "Turn Left", leading to a restaurant.

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 process algebras that exist are designed to handle a lot more complex situations such as unobservable actions, commonly referred to as τ-actions, recursion, which lets a process repeat itself or other processes, and deadlock, which is a state where no desirable outcomes can be reached.

There are many process algebras that exist, the most famous and seminal being CSP (Brookes et al., 1984), CCS (Milner, 1980), and ACP (Bergstra and Klop, 1984), (Bergstra and Klop, 1989), with some other popular calculi being the π-calculus and its various extensions (Milner et al., 1992), (Parrow and Victor, 1998), (Abadi and Gordon, 1999) which have been used to varying degrees in fields like Biology, Business, and Cryptography, or the Ambient Calculus (Cardelli and Gordon, 1998) 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.(Parrow, 2008) 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 (Gorla, 2010). 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 (Peters, 2019). 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 (Gorla, 2010), (van Glabbeek, 2018).

2.3 CSP

CSP (Communicating Sequential Processes) (Brookes et al., 1984) 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) is a Process Algebra developed by Jan Bergstra and Jan Willem Klop (Bergstra and Klop, 1984). 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.

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. ACP_τ (Bergstra and Klop, 1989) is an extension of ACP that includes an extra action τ which is used to represent actions that are unobservable, or changeable, from a human perspective.

[WIP]

Chapter 3

A formal definition of CSP and ACP_F^τ

3.1 Languages and Expressiveness

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

Definition 3.1: Languages

Via van Glabbeek (2018), 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 $\mathscr{D}$.

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 van Glabbeek (2018), a **translation** from a language $\mathscr{L}$ to a language $\mathscr{L}'$ is a mapping $\mathscr{T} : \mathbb{T}_{\mathscr{L}} \to \mathbb{T}_{\mathscr{L}'}$

The preferred way to measure expressiveness is via relative expressiveness, compared to Absolute expressiveness. Via Parrow (2008), 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 van Glabbeek (2018), 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 van Glabbeek (2018), 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 Parrow (2008) 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 van Glabbeek (2018), 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 $E_1,\ldots,E_n \in \mathbb{T}_{\mathscr{L}}$

Here, the set $\mathscr{X}$ is the set of process variables, such as specified in the recursive specification. We should also specify that $\mathscr{T}(a) = a$ for each $a \in A$ specifies that any actions are consistent between algebras.

3.2 ACP_F^τ

3.2.1 Basic ACP_τ

Definition 3.6: ACP_τ

The process algebra ACP_τ as described in Bergstra and Klop (1989) is parameterised by a set of actions, Σ, and a communication function $|$. The grammar of ACP_τ can be described with the following operations:

$$P,Q ::= a \mid \delta \mid E.F \mid E+F \mid E \,||\, F \mid E \,⌊\!⌊\, F \mid E|F \mid \partial_H(E) \mid \tau_I(P) \mid$$

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 Σ_τ to be the set $\Sigma \cup \{\tau\}$, as the silent step is not included in Σ. The operations can be described in the following manner:

- **Action**, or a, is any action.
- **Deadlock**, or δ, 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 τ.

3.2.2 Communication and Merge

The operations Merge ($P\,||\,Q$), Left Merge ($P \mathbin{\lfloor\!\lfloor} Q$), 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

$$P\,||\,Q = P \mathbin{\lfloor\!\lfloor} Q + Q \mathbin{\lfloor\!\lfloor} P + P \mid Q$$

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

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

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 $P \mid Q$.

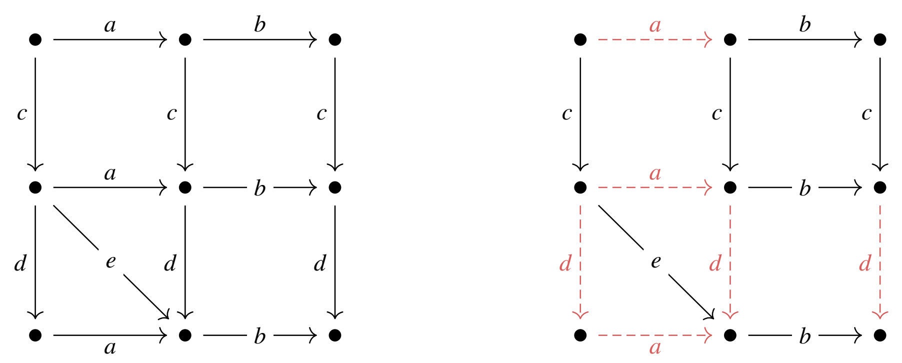
The operation $\mid$ is a function $A \times A \rightharpoonup A$ that defines valid communications between the two processes[1]. This can be thought of as a hand-shaking action between P and Q. Together with the restriction operator, communications can now be formed between two processes. An example 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 δ, is achieved. This results in twice as many Operational Rules, which we will see in Section 3.4, 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 ACP_ε, which adds a successful termination action at the cost of extending process graphs with more

[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`.

actions and the loss of a true Strong Bisimulation due to the extra action ε, and the fragment aprACP, which uses Action Prefixing similar to CSP or the language CCS at the cost of losing asymmetry and compatibilityGaravel (2015). 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 ACP_F^τ

A proposed extension of ACP adds a Functional Renaming operator, as shown in van Glabbeek (1995), 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^τ. Our final grammar for the language ACP_F^τ is formally defined below.

Definition 3.7: ACP_F^τ

The Algebra ACP_F^τ as described in Bergstra and Klop (1989), van Glabbeek (1995), is parameterised by a set of actions, Σ, and a communication function $|$. The grammar of ACP_F^τ can be described with the following operations:

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

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 [REF] is parameterised on a set of communications Σ, and the grammar consists of the operations:

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

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 does not disappear. This differs from ACP Alternative Composition, as in ACP a τ 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, e.g. a and a compared to the synchronisable actions being defined with an operator like $|$.
- **Concealment**, or $P \backslash Q$, is an operation that removes actions from a set Q, acting similarly to the Abstraction operator τ_I of ACP.j
- **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 [REF] are a method of describing the actions of a process. 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 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.[2]

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

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

where the meaning of the proof tables is:

- If P can perform an action α to P', then $P\,||\,Q$ can perform an action α 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 α to Q', then $P\,||\,Q$ can perform an action α to $P\,||\,Q'$

The rest of the operations of ACP_F^τ and CSP can also be represented similarly in the following GSOS tables:

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 Baeten and Weijland (1990):

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

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

Table 3.1: Structural operational semantics of CSP

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 $s'Rt'$.

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 ACP_F^τ 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[3]. 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 b action, and a c action. Clearly, this implies that *Bisimulation*

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

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

Table 3.2: Structural operational semantics of the language ACP^{τ}_{F}. The operational semantics for Successful Termination in the operators *Alternative Composition*, *Merge*, *Left Merge*, *Communication Merge*, *Abstraction*, *Encapsulation*, *Recursion*, *Encapsulation*, and *Functional Renaming* are omitted. A full list of operational rules can be found in Table B.1

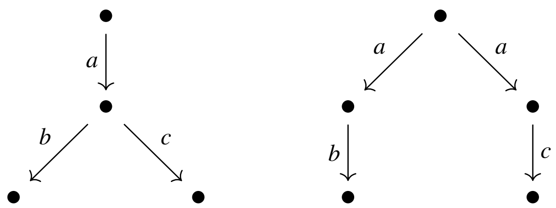


Figure 3.2: The process $a.(b+c)$ (left), compared to the process $a.b+a.c$ (right). Both Processes have the same Traces - $\{a.b, a.c\}$, however they are not Strongly Bisimilar

.

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 equivalent if they can move the same way when looking at **External actions**. In particular, this excludes internal actions, τ, 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, which adds the condition that the first action in the process must be Strongly Bisimilar, which leads to Rooted Branching Bisimilarity being a Congruence (Fokkink, 2000). This means that Rooted Branching Bisimilarity is not only finer than Branching Bisimilarity, 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 Baeten and Weijland (1990):

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'
3. 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** 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 $s'Rt'$

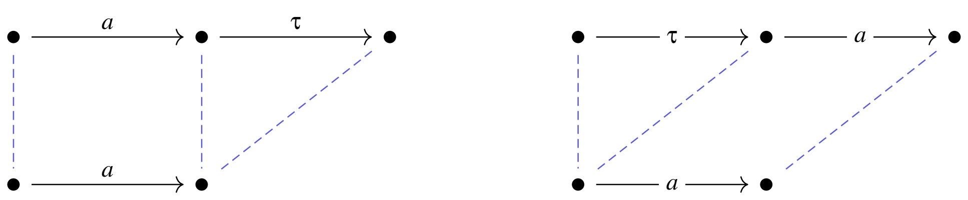


Figure 3.3: Example of Branching Bisimilarity. The process on the left ($a.\tau$ compared to a) is Rooted Branching Bisimilar, and the process on the right ($\tau.a$ compared to a) is only Branching Bisimilar. A blue dotted line indicates a relation between states

Chapter 4

A Translation of CSP to ACP_F^τ

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.

$$\begin{aligned}\mathscr{T}(STOP) &= \delta \\ \mathscr{T}(a \to P) &= a.\mathscr{T}(P) \\ \mathscr{T}(P \backslash A) &= \tau_A \mathscr{T}(P)\end{aligned}$$

4.2 Trivial Translations

- **Divergence** is the process that diverges via infinite internal actions. It is defined by the following rule:

  $$\text{div} \xrightarrow{\tau} \text{div}$$

  and then can be directly translated via recursion in ACP in the following rule:

  $$\mathscr{T}(\text{div}) = \langle X \mid X = \tau.X \rangle$$

- **Renaming** is an operation that renames actions in processes according to a function. There is no equivalent function in plain ACP_τ, with the closest operation being $\tau_I(P)$ which abstracts actions in I to internal actions. This is possible in ACP_F^τ, and in fact our translation is trivially

  $$\mathscr{T}(f(P)) = f(\mathscr{T}(P))$$

- **Internal Choice** is an operation that emulates a choice of actions that cannot be decided by the user. CSP in particular differs from ACP in that external choice and internal choice are separate operations, while in ACP, the alternative choice operator $+$ handles choice, albeit slightly differently. With the internal choice operator τ, a translation for CSP Internal choice into ACP is easily written as

  $$\mathscr{T}(P \sqcap Q) = \tau.\mathscr{T}(P) + \tau.\mathscr{T}(Q)$$

The above translations are all valid up to Strong Bisimilarity. The other operators are slightly harder to translate.

4.3 Helper Operators for the language ACP_F^τ

4.3.1 Subsets of Σ

Working in the language ACP_τ with the extension of Functional Renaming (written ACP_F^τ), we first recall that ACP_F^τ is paramaterised by a set of actions Σ as defined in 3.2. Also recall that Σ_τ is defined as the set $\Sigma \cup \{\tau\}$. We then start by defining some subsets of Σ which we will use in our encodings.

Definition 4.1: Subsets of A

The set $\Sigma \in \mathbb{T}$ is the set of all actions.

- $\Sigma_0 \subseteq A$ is the set of actions that actually get used in processes
- $A \subseteq \Sigma$ is a set of target actions. This is used in operators such as CSP Parallel Composition, which only communicates over a set.
- $H_0 = \Sigma - \Sigma_0$ is the set of working space operators, or any other action that doesn't get used
- $\mathscr{H} \subseteq H_0$ 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 $\mathscr{H}$

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

4.3.2 Triggering

We define an operator $\Gamma(P)$ that emulates the Triggering operator of MEIJE [REFER]. For a trace $a.b.\cdots$ on a process P, the triggering operator can be represented as an operator that tags the first action of a process.

First, we define a function $f_{\mathtt{trig}}$ and communications for the operations `first` and `next`.

Def 4.2: Communications

Define communications where:

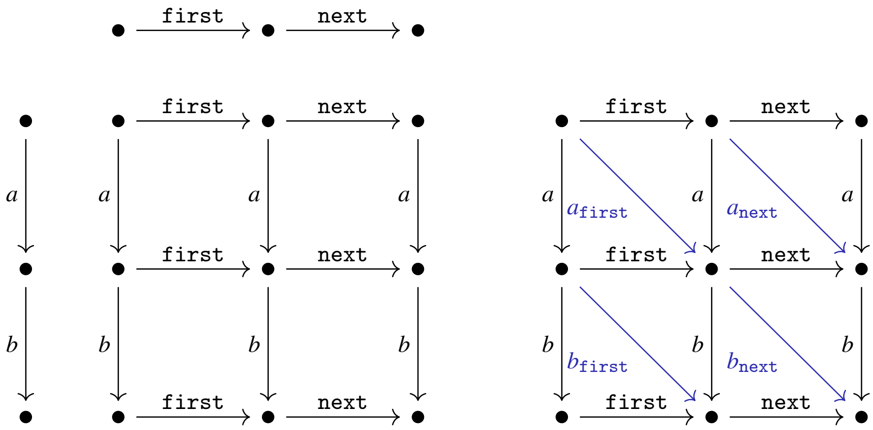
$$a|\mathtt{first} = a_{\mathtt{first}}$$
$$a|\mathtt{next} = a_{\mathtt{next}}$$

Def 4.3: Triggering Function

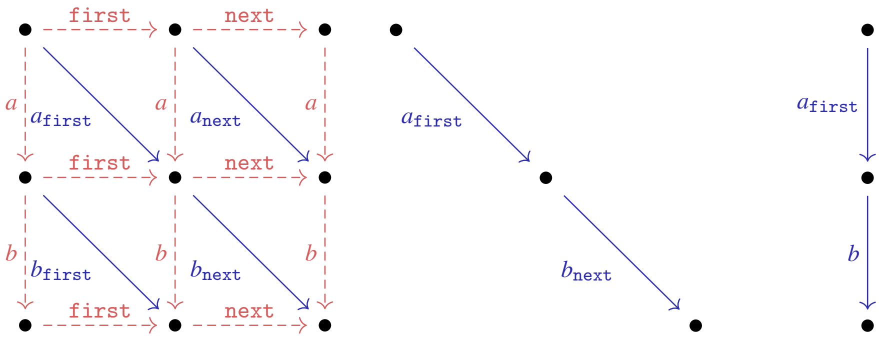
Define $f_{\mathtt{trig}} : \Sigma_\tau \to \Sigma_\tau$ where:

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

We use the notation of a^∞ as syntactic sugar to mean $\langle X \mid X = a.X \rangle$. Using the sets



(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 $P = a.b.c$

defined in 4.1, we can now define $\Gamma(P)$ as such:

Definition 4.4: Triggering in ACP

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

is an operator that turns a trace of a process P, $a.b.c....$ into the trace

$$a_{\texttt{ini}}.b.c....$$

This works in the following method:

a) Merge the process P with the process $\texttt{first}.\texttt{next}.\texttt{next}....$ Via Def 4.2, this will produce a lattice of P and $\texttt{first}.(\texttt{next}^{\infty})$, with communications on every square, but most importantly, a chain of communications going down the centre of the form.

$$a_{\texttt{first}}.b_{\texttt{next}}.c_{\texttt{next}}\cdots \tag{4.1}$$

b) Restrict the actions in H_1. Since all the actions in $P, \texttt{first}.(\texttt{next}^{\infty}) \in H_1$ this effectively restricts both sides of the left merge, leaving only communications from the initial state. This leaves equation 4.1 as the only remaining trace.

c) Apply $f_{\mathtt{trig}}$ to equation 4.1. Via 4.3, the final result is

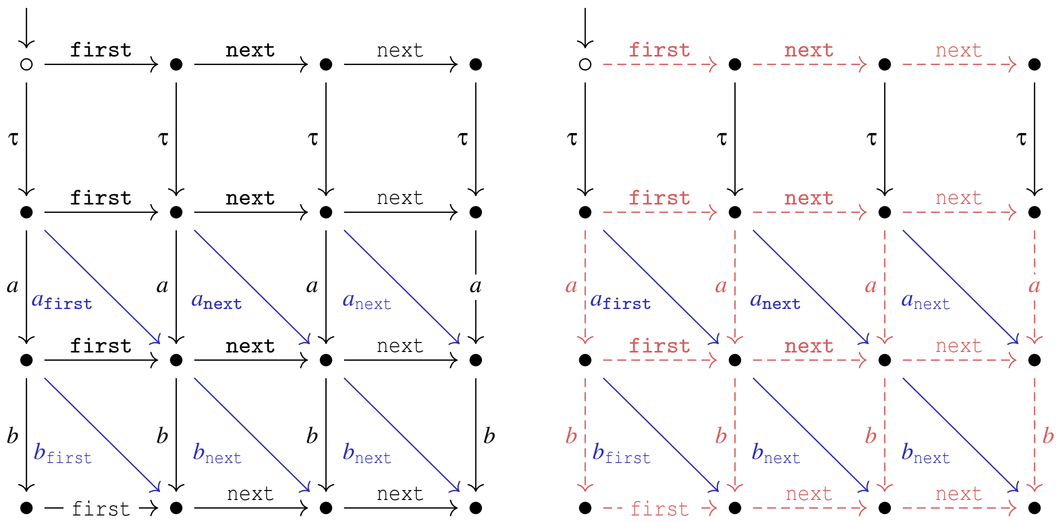
$$a_{\mathtt{ini}}.b.c\ldots \tag{4.2}$$

The process is now exactly as stated in Definition 4.4.

Note that since $\tau \notin A$, ∂_{H_1} will not restrict τ, and additionally since τ does not communicate with any actions, Step 2 effectively becomes any amount of τ steps followed by the diagonal trace immediately following that. This results in cases $\Gamma(P)$ where $P = \tau.b.c\ldots$ becoming the trace

$$\tau.b_{\mathtt{ini}}.c\ldots.$$

effectively skipping τ's, then acting the same as processes that don't start with a τ.



(a) Left merge with τ communications (b) Restriction of H_1 with τ actions

Figure 4.2: Example of Triggering operator with internal actions

4.3.3 Associativity and Postfix Function

From the axioms of ACP_τ (Bergstra and Klop, 1989), we have the following axioms of associativity with the communication operator $|$ shown in the table below:

Communication function in ACP_τ
$a\|b = b\|a$
$(a\|b)\|c = a\|(b\|c)$
$\delta\|a = \delta$

Table 4.1: Axioms of ACP_τ Communication

Bearing the above axioms in mind, we have the potential to run into problems with this with our communications. For example, in the proposed translation for the External Choice operator $\square$ (4.4.3), a simplified version of the translation would have the following communications:

$$a|\mathtt{first} = a_{\mathtt{first}} \qquad a|\mathtt{next} = a_{\mathtt{next}} \qquad a_{\mathtt{first}}|\mathtt{choose} = a$$

This might work at first glance, but the Associativity axiom does not hold true in this case, such as in the following counterexample:

$$\begin{aligned} a|\mathtt{first}|\mathtt{choose} &= (a|\mathtt{first})|\mathtt{choose} = a_{\mathtt{first}}|\mathtt{choose} = a \\ &= a|(\mathtt{first}|\mathtt{choose}) = a|\delta = \delta \end{aligned}$$

From this example, it is clear that the proposed communications would not satisfy the axioms of ACP_τ. It is for this reason that in 4.3, we have the rule

$$f_{\mathtt{trig}}(a_{\mathtt{first}}) = a_{\mathtt{ini}} \qquad \text{instead of} \qquad f_{\mathtt{trig}}(a_{\mathtt{first}}) = a_{\mathtt{first}}$$

A preferred communications function is as follows:

$$a|\mathtt{first} = a_{\mathtt{first}} \qquad a|\mathtt{next} = a_{\mathtt{next}} \qquad a_{\mathtt{ini}}|\mathtt{choose} = a$$

$$\begin{aligned} a|\mathtt{first}|\mathtt{choose} &= (a|\mathtt{first})|\mathtt{choose} = a_{\mathtt{first}}|\mathtt{choose} = \delta \\ &= a|(\mathtt{first}|\mathtt{choose}) = a|\delta = \delta \end{aligned}$$

It is important to note that this scenario would never actually occur in practice, since $\Gamma(P)$ takes precedence and hence `first|choose` would never happen, but the communication function must work over every action regardless of whether it will get used in practice.

Lemma 4.5: Keeping Associativity in Communications

For three actions $a,b,c \in A$, if $a|b = c$, then if c does not appear on the left-hand side of any other communication, Associativity will be satisfied.

Proof. WIP. QED ∎

We define a compatibility function to prevent issues with associativity. Via 4.5, define a tag $a_{\texttt{post}}$ which will act as c for all our communications, which satisfies the Lemma. Then, our final step will be to rename a_{post} back to a for any affected actions. This works in the following way:

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

Definition 4.6: Postfix function

Let $f_{\text{post}} : \Sigma_\tau \to \Sigma_\tau$ where

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

4.4 Translations for the remaining CSP Operators

4.4.1 Communications and Functional Renaming

We define communications for our translation in addition to the ones previously defined for our helper functions. Also note that these functions are defined over Σ_τ for bookkeeping purposes as internal actions τ cannot get renamed and therefore stay as an internal action no matter the function. These are defined as follows:

Definition 4.7: Helper Functions

In addition to the function $f_{\texttt{trig}}$ defined in 4.3, and the postfix function defined in 4.6:

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

We define functions for the remaining operators below. We use the notation A_T to signify a target set, as used in 4.4.2, 4.4.5, and λ to signify actions in A

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

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

2. $f_{\texttt{origin}} : \Sigma_\tau \to \Sigma_\tau$ is a function that renames actions in a process for use in operators. This is used in the translation of the Interrupt operator (4.4.4)

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

3. $f_{\texttt{split}} : \Sigma_\tau \to \Sigma_\tau$ 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 (4.4.5)

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

Definition 4.8: Communications

In addition to the communications for the Triggering operator defined in 4.2

$$a|\texttt{first} = a_{\texttt{first}} \qquad a|\texttt{next} = a_{\texttt{next}}$$

we define additional communications for the functions defined above:

1. $a_{\texttt{syn}}|a_{\texttt{syn}} = a_{\texttt{post}}$. This is used in the translation of Parallel Composition (4.4.2)
2. $a_{\texttt{ini}}|\texttt{choose} = a_{\texttt{post}}$. This is used in the translation of External choice (4.4.3)
3. $a_{\texttt{origin}}|\texttt{origin} = a_{\texttt{post}}$. This is used in the translation of Interrupt and Throw operator (4.4.4, 4.4.5)
4. $a_{\texttt{ini}}|\texttt{split} = a_{\texttt{post}}$. This is used in the translation of the Interrupt operator (4.4.4)
5. $a_{\texttt{split}}|\texttt{split} = a_{\texttt{post}}$. This is used in the translation of the Throw operator (4.4.5)

4.4.2 Parallel Composition

The parallel composition $||_A$ is defined with the following rules:

$$\frac{P \xrightarrow{\alpha} P' \quad (\alpha \notin A)}{P\,||_A\,Q \xrightarrow{\alpha} P'\,||_A\,Q} \qquad \frac{Q \xrightarrow{\alpha} Q' \quad (\alpha \notin A)}{P\,||_A\,Q \xrightarrow{a} P\,||_A\,Q'} \qquad \frac{P \xrightarrow{a} P'\; Q \xrightarrow{a} Q' (\alpha \in A)}{P\,||_A\,Q \xrightarrow{a} P'\,||_A\,Q'} \tag{4.3}$$

In other words, a left merge action can be taken by all actions of P and Q, which is the same as the ACP_F^τ equivalent of parallel composition with the one difference being that in CSP, the action must be the same in P and Q, whereas in ACP_F^τ the action is defined with a communication 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 identical marked actions. We can do this via the following functions and communications:

Definition 4.9: Functions and Communications - Parallel Composition

As defined in 4.7 and 4.8, the following communications and functions are defined over Parallel Composition:

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

$$a_{\texttt{syn}}|a_{\texttt{syn}} = a_{\texttt{post}}$$

A translation for Parallel Composition can then be written as the following:

$$\mathscr{T}(P\,||_A\,Q) = \partial_{H_0}(f_{\text{post}}(f_{\text{syn}}(\mathscr{T}(P))\,||\,f_{\text{syn}}(\mathscr{T}(Q))))$$

4.4.3 External choice

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

$$\frac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P \Box Q \xrightarrow{a} Q'} \qquad \frac{P \xrightarrow{\tau} P'}{P \Box Q \xrightarrow{\tau} P' \Box Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \Box Q \xrightarrow{a} P \Box Q'} \quad (4.4)$$

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 ACP_F^τ Alternative Choice operator $(+)$, as $+$ will not let you select externally if an internal action is made. For our encoding, we take $\mathscr{H} = \{\texttt{first}, \texttt{next}, \texttt{choose}\}$, and therefore $H_1 = A_0 \uplus \{\texttt{first}, \texttt{next}, \texttt{choose}\}$ as defined in Definition 4.1.

Definition 4.10: Functions and Communications - External Choice

As defined in 4.7 and 4.8, the following communications and functions are defined over External Choice:

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

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

Additionally, recall that the Triggering operator (4.4) is defined as:

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

We can then define an encoding of the CSP external choice operator $\Box$ in ACP_F^τ in the following equation

$$\mathscr{T}(P \Box Q) = \partial_{H_0}\Big(f_2\Big[\Gamma[\mathscr{T}(P)] \,||\, \texttt{choose} \,||\, \Gamma[\mathscr{T}(Q)]\Big]\Big)$$

4.4.4 Interrupt

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

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

In other words, we can take an external action from P without fulfilling the operator, 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 $H_1 = A_0 \uplus \{\texttt{first}, \texttt{next}, \texttt{origin}, \texttt{split}\}$ as defined in 4.1.

Definition 4.11: Functions and Communications - Interrupt

As defined in 4.7 and 4.8, the following communications and functions are defined for our translation:q

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

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

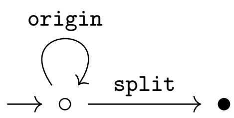
Additionally, recall that the Triggering operator (4.4) is defined as:

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

We can now define an encoding of the CSP Interrupt operator $\triangle$ in ACP_F^τ. We start off with a new hoper process, which we will call Π. This is defined as the recursive equation

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

Or, visualised as a process graph:



From this, an encoding can be written in the following way:

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

4.4.5 Throw

The Throw operator Θ_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, which we will call A_T. However, the moment an action in A_T is made, The process then diverts to Q. This can be also throught as an error checking operator. Similarly to the interrupt operator, for our encoding, we take

$$\mathscr{H} = \{\texttt{first}, \texttt{next}, \texttt{origin}, \texttt{split}\}$$

, and therefore $H_1 = A_0 \uplus \{\texttt{first}, \texttt{next}, \texttt{origin}, \texttt{split}\}$ as defined in 4.1.

Definition 4.12: Functions and Communications - Throw

As defined in 4.7 and 4.8, the following communications and functions are defined for our translation:

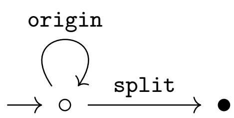
$$f_{\texttt{post}}(\alpha) = \begin{cases} a & \text{if } \alpha = a_{\text{post}} \\ \alpha & \text{otherwise} \end{cases} \qquad f_{\texttt{split}}(\alpha) = \begin{cases} a_{\texttt{split}} & \text{if } a \in A \\ a_{\texttt{origin}} & \text{if } a \notin A,\ a \in \Sigma \\ \tau & \text{otherwise}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

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

We can now define an encoding of the CSP Throw operator Θ_A in ACP$^{\tau}_{F}$. We employ the use of the same helper process as in the translation of the Interrupt operator, Π which is defined as

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

Or, visualised as a process graph:



From this, an encoding can be written in the following way:

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

4.4.6 Sliding Choice

The sliding choice, or sliding operator $\triangleright$ is defined with the following rules:

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

In other words, this operator lets you take an external action on P, however there is a second process that may at any point before the external action taken in P “time out” and move to Q instead.

WIP

4.4.7 Final Translation

Recall the grammar of the languages CSP and ACP_F^τ as defined in Definitions

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

Recall from Definitions 4.7 and 4.8 the list of helper functions and communications:

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

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

We now define our 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 ACP_F^τ. We define a translation $\mathscr{T} : \mathbb{T}_{\mathrm{CSP}} \to \mathbb{T}_{\mathrm{ACP}_F^\tau}$ defined as such:

$$\begin{aligned}
\mathscr{T}(STOP) &= \delta \\
\mathscr{T}(\mathrm{div}) &= \langle X \mid X = \tau.X\rangle \\
\mathscr{T}(a \to P) &= a.\mathscr{T}(P) \\
\mathscr{T}(P \sqcap Q) &= \tau.\mathscr{T}(P) + \tau.\mathscr{T}(Q) \\
\mathscr{T}(P \Box Q) &= \partial_{H_0}\left(f_{\mathtt{post}}\left[\Gamma[\mathscr{T}(P)] \mid\mid \mathtt{choose} \mid\mid \Gamma[\mathscr{T}(Q)]\right]\right) \\
\mathscr{T}(P ||_A Q) &= \partial_{H_0}\left(f_{\mathtt{post}}\left[f_{\mathrm{syn}}(\mathscr{T}(P)) \mid\mid f_{\mathrm{syn}}(\mathscr{T}(Q))\right]\right) \\
\mathscr{T}(P \backslash A) &= \partial_A \mathscr{T}(P) \\
\mathscr{T}(f(P)) &= f(\mathscr{T}(P)) \\
\mathscr{T}(P \triangle Q) &= \partial_{H_0}\left(f_{\mathtt{post}}\left[((f_{\mathtt{origin}}\mathscr{T}(P)) \mid\mid \Pi) \mid\mid \Gamma(\mathscr{T}(Q))\right]\right) \\
\mathscr{T}(P \Theta_A Q) &= \partial_{H_0}\left(f_{\mathtt{post}}\left[(f_{\mathtt{split}}\mathscr{T}(P)) || \Pi.\mathscr{T}(Q)\right]\right)
\end{aligned}$$

Chapter 5

Validity of the Encoding

Referring back to our translation 4.13, the translation for External Choice

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

has identical behaviour to $P+Q$ when dealing with processes with only external actions. However, on processes with internal actions, the translation is not so trivial. The addition of an internal action works in the translation's favour for actions such as deferring a `start` tag to the first visible action (see 4.2), but it can also backfire, as the translation largely relies on removing unwanted left-merges and communications through restriction operators. As internal actions are non-interactable it means restrictions that *should* remove all remaining actions will end up with unwanted left-over τ moves.
For example, if we take the process $a \Box \tau.b \in$ CSP, the resulting translation in ACP^{τ}_{F} is

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

which has the following process graph:

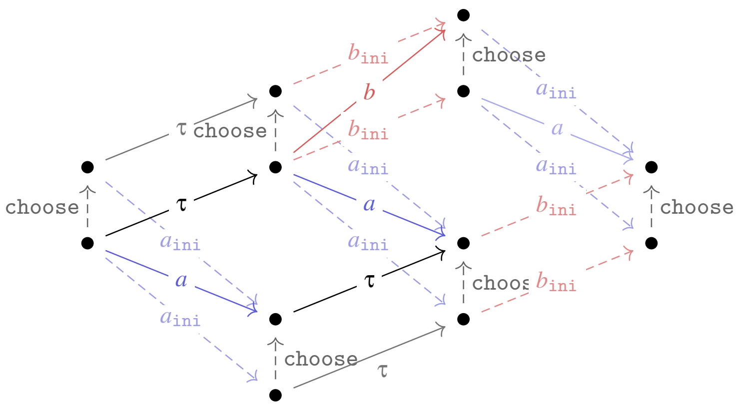


Figure 5.1: Counterexample for Strong Bisimilarity with the processes $P=a$ and $Q=\tau.b$. The result of the translation is $a.\tau+\tau.(a+b) \not\Leftrightarrow a+\tau.(a+b)$

5.1 Prerequisites and Helper Theorems

5.1.1 Rooted Branching Bisimilarity

Recall from Definition 3.10 the definition of Branching and Rooted Branching Bisimilarity:

A relation 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'
3. 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** 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 a s' with $\text{root}(P) \xrightarrow{\alpha} s'$ and $s'Rt'$

5.1.2 Lemmas and Theorems

Lemma 5.1

For a trace in a process $P \in \text{ACP}_F^\tau$, if there exists P' such that $P \xrightarrow{\tau} P'$ then application of the Triggering operator shown in 4.4 can be applied as $\Gamma[P] \xrightarrow{\tau} \Gamma[P']$. Alternatively,

$$\Gamma[P] = \tau.\Gamma[P']$$

Proof. We have that $P = \tau.P'$ and therefore $\Gamma[P] = \Gamma[\tau.P']$. However, since internal actions cannot interact with anything inside the triggering operator, the resulting function is the same as if the τ was instead outside the function. A diagram is provided in 4.2. □

Lemma 5.2

For a process $P \in \text{ACP}_F^\tau$, if there exists P' such that $P \xrightarrow{a} P'$ then for any action $a \in A_0$ we have

$$(\Gamma[P] \,||\, \texttt{choose}) \xrightarrow{a} P' \quad \text{and} \quad (\texttt{choose} \,||\, (\Gamma[P]) \xrightarrow{a} P'$$

Proof. Directly follows from communications □

Lemma 5.3

For processes $a.P \in \text{ACP}_F^\tau$ where $a \in A_0$, if $\exists b.Q$ where $b \in A_0$ we have

$$\partial_{H_0}(b.Q \,||\, \Gamma[a.P]) = b.Q$$

Alternatively, for processes $\tau^*.P \in \text{ACP}_F^\tau$, where τ^* indicates a chain of consecutive τ, possibly 0, if there exists a process $b.Q$ where $b \in A_0$ we have

$$\partial_{H_0}(b.Q \,||\, \tau^*.\Gamma[P]) = b.Q \,||\, \tau^*$$

Proof. The Triggering operator Γ turns a trace into a renamed trace of the form

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

The process $a_{\mathtt{ini}}$ does not communicate with anything other than `choose` which is not in A_0. Therefore, the restriction operator will remove all the Γ left merges, leaving $b.Q$. However, if there are internal actions that cannot interact with restrictions and communications, then the final process will be a communication with any remaining internal actions before the first cut off external action. □

Lemma 5.4

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 τ^* indicating a chain of τ, possibly 0.

Proof. just works

WIP □

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 bisimulation relation.

Definition 5.5: 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 ACP_F^τ. 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 External choice

From Definitions 4.7, 4.8, and 4.13, the translation of external choice is:

$$\mathscr{T}(P \Box Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma[\mathscr{T}(P)] \mid\mid_A \texttt{choose} \mid\mid_A \Gamma[\mathscr{T}(Q)]\Big]\Big)$$

Using the following communications and functions:

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

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

Proof. Let $P, Q \in \text{CSP}$ be two processes. We want to show that $P \Box Q =_{RBB} \mathscr{T}(P \Box Q)$. i.e.: we want to show that any move will result in a process that satisfies RBB. We show this by exhausting all possible moves that $P \Box Q$ can take, and confirm that $\mathscr{T}(P \Box Q)$ can also take them, up to Rooted Branching Bisimilarity.

Case 1: a action on P. Let $a \in A_0$ and P' such that $P \xrightarrow{a} P'$. In the domain of CSP, this results in the process

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

Now working in ACP_F^τ, we want to show that the translation is valid up to RBB, i.e. $\mathscr{T}(P') =_{RBB} P'$. Via Lemma 5.2, we can derive the following equation

$$\mathscr{T}(P \Box Q) = \partial_{H_0}\Big(f_{\texttt{post}}\Big[\Gamma[\mathscr{T}(P)] \mid\mid \texttt{choose} \mid\mid \Gamma[\mathscr{T}(Q)]\Big]\Big) \xrightarrow{a}$$
$$\partial_{H_0}\Big(f_{\texttt{post}}\Big[\mathscr{T}(P') \mid\mid \Gamma[\mathscr{T}(Q)]\Big]\Big)$$

This is not yet a process that is comparable to P', so we look at the next step. Due to the Triggering operator Γ 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 τ actions behind it. Due to the restriction of H_0, since a `ini`-tagged operator can only communicate with the action `choose`, any of the actions past τ^* will get restricted, leaving

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

for every trace Q_c in Q. Via Lemma 5.4, 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 worst Branching Bisimilar, and therefore as the first action is related Strongly the process is valid up to Rooted Branching Bisimilarity when taking an a action on P.

Case 2: τ action on P. Let P' such that $P \xrightarrow{\tau} P'$. In the domain of CSP, this results in the process

$$P \Box Q \xrightarrow{\tau} P' \Box Q$$

Now working in ACP_F^τ, we want to show that the translation is valid up to RBB, i.e. $\mathscr{T}(P' \,\Box\, Q) =_{RBB} P' \,\Box\, Q$. Via Lemma 5.1, we can now derive the following equation

$$\partial_{H_0}\Big(f_1\Big[\Gamma[\mathscr{T}(P)]\,||\,\mathtt{choose}\,||\,\Gamma[\mathscr{T}(Q)]\Big]\Big) \xrightarrow{\tau}$$
$$\partial_{H_0}\Big(f_1\Big[\Gamma[\mathscr{T}(P')]\,||\,\mathtt{choose}\,||\,\Gamma[\mathscr{T}(Q)]\Big]\Big)$$

Which is strongly bisimilar to $P' \,\Box\, Q$, therefore a τ action on P is also valid up to Rooted Branching Bisimilarity.

Case 3, Case 4: The same logic from option 1 and option 2 can be applied to Q and P to 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 Rooted Branching Bisimilar, and therefore Compositional. □

5.2.2 Interrupt

Similarly to external choice, this is also Rooted Branching Bisimilar. We prove this similarly to the previous operator. From Definitions 4.7, 4.8, and 4.13, we recall our translation of the Interrupt operator:

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

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

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

Proof. Let $P, Q \in$ CSP be two processes. We want to show $P \triangle Q =_{RBB} \mathscr{T}(P \triangle Q)$. i.e.: we want to show that any move will result in a process that satisfies RBB. We show this by exhausting all possible moves that $P \triangle Q$ can take, and confirm that $\mathscr{T}(P \triangle Q)$ can also take them, up to Rooted Branching Bisimilarity.

□

5.2.3 Generalising

Lemma 5.6: Maybe?

I claim that a Strong Bisimulation cannot occur and RBB is the finest equivalence able to be translated

Theorem 5.7: Maybe 2

The following diagram cannot be modelled in ACP using Communication of two processes a and τ

$$\begin{tikzcd}
\bullet \arrow[r, "\tau"] \arrow[d, "a"'] & \bullet \\
\bullet &
\end{tikzcd}$$

Proof. $\square$

Chapter 6

Results

6.1 Results

From the results in this paper, I have shown a translation from the language CSP to the language ACP_F^τ. I have shown that this translation is valid up to Rooted Branching Bisimulation for operators excluding the CSP Sliding choice operator.

6.2 Limitations

Not Strongly Bisimilar.

Bibliography

Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, January 1999. ISSN 0890-5401. doi: 10.1006/inco.1998.2740.

J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335 (2):131–146, May 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2004.07.036.

J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1990. doi: 10.1017/CBO9780511624193.

J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109–137, January 1984. ISSN 0019-9958. doi: 10.1016/S0019-9958(84)80025-X.

J. A. Bergstra and J. W. Klop. ACPτ a universal axiom system for process specification. In Martin Wirsing and Jan A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, pages 445–463, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. ISBN 978-3-540-46758-8.

S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3):560–599, June 1984. ISSN 0004-5411. doi: 10.1145/828.833.

Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structures*, pages 140–155, Berlin, Heidelberg, 1998. Springer. ISBN 978-3-540-69720-6. doi: 10.1007/BFb0053547.

Wan Fokkink. Rooted Branching Bisimulation as a Congruence. *Journal of Computer and System Sciences*, 60(1):13–37, February 2000. ISSN 0022-0000. doi: 10.1006/jcss.1999.1663.

Hubert Garavel. Revisiting sequential composition in process calculi. *Journal of Logical and Algebraic Methods in Programming*, 84(6):742–762, 2015. ISSN 2352-2208. doi: 10.1016/j.jlamp.2015.08.001.

Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, September 2010. ISSN 0890-5401. doi: 10.1016/j.ic.2010.05.002.

Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 1980. ISBN 978-3-540-10235-9 978-3-540-38311-6. doi: 10.1007/3-540-10235-3.

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992. ISSN 0890-5401. doi: 10.1016/0890-5401(92)90008-4.

J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.98CB36226)*, pages 176–185, June 1998. doi: 10.1109/LICS.1998.705654.

Joachim Parrow. Expressiveness of Process Algebras. *Electronic Notes in Theoretical Computer Science*, 209:173–186, April 2008. ISSN 1571-0661. doi: 10.1016/j.entcs.2008.04.011.

Kirstin Peters. Comparing Process Calculi Using Encodings. *Electronic Proceedings in Theoretical Computer Science*, 300:19–38, August 2019. ISSN 2075-2180. doi: 10.4204/EPTCS.300.2.

Rob van Glabbeek. On the Expressiveness of ACP. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes*, pages 188–217, London, 1995. Springer London. ISBN 978-1-4471-2120-6.

Rob van Glabbeek. A theory of encodings and expressiveness (extended abstract) - (extended abstract). In Christel Baier and Ugo Dal Lago, editors, *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*, volume 10803 of *Lecture Notes in Computer Science*, pages 183–202. Springer, 2018. doi: 10.1007/978-3-319-89366-2_10.

Appendix A

Diagrams

A.1 Examples of Translations

In this section I will provide visual aids for the encoding of translations of examples of operators

A.2 Throw

$$
\begin{tikzcd}
\bullet \arrow[r, dashed, "\mathtt{origin}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] \arrow[dr, "a"] & \bullet \arrow[r, dashed, "\mathtt{origin}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] \arrow[dr, "a"] & \bullet \arrow[r, dashed, "\mathtt{split}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] & \bullet \arrow[r, "q_1"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] & \bullet \arrow[d, dashed, "a_{\mathtt{origin}}"'] \\
\bullet \arrow[r, dashed, "\mathtt{origin}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] \arrow[dr, "a"] & \bullet \arrow[r, dashed, "\mathtt{origin}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] \arrow[dr, "a"] & \bullet \arrow[r, dashed, "\mathtt{split}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] & \bullet \arrow[r, "q_1"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] & \bullet \arrow[d, dashed, "a_{\mathtt{origin}}"'] \\
\bullet \arrow[r, dashed, "\mathtt{origin}"] \arrow[d, dashed, "b_{\mathtt{split}}"'] & \bullet \arrow[r, dashed, "\mathtt{origin}"] \arrow[d, dashed, "b_{\mathtt{split}}"'] & \bullet \arrow[r, dashed, "\mathtt{split}"] \arrow[d, dashed, "b_{\mathtt{split}}"'] \arrow[dr, "b"] & \bullet \arrow[r, "q_1"] \arrow[d, dashed, "b_{\mathtt{split}}"'] & \bullet \arrow[d, dashed, "b_{\mathtt{split}}"'] \\
\bullet \arrow[r, dashed, "\mathtt{origin}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] \arrow[dr, "a"] & \bullet \arrow[r, dashed, "\mathtt{origin}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] \arrow[dr, "a"] & \bullet \arrow[r, dashed, "\mathtt{split}"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] & \bullet \arrow[r, "q_1"] \arrow[d, dashed, "a_{\mathtt{origin}}"'] & \bullet \arrow[d, dashed, "a_{\mathtt{origin}}"'] \\
\bullet \arrow[r, dashed, "\mathtt{origin}"] & \bullet \arrow[r, dashed, "\mathtt{origin}"] & \bullet \arrow[r, dashed, "\mathtt{split}"] & \bullet \arrow[r, "q_1"] & \bullet
\end{tikzcd}
$$

A.3 Counterexamples to Translations

In this section I will provide visual aid for counterexamples of operators that are not valid up to Strong Bisimularity.

A.3.1 Interrupt Operator

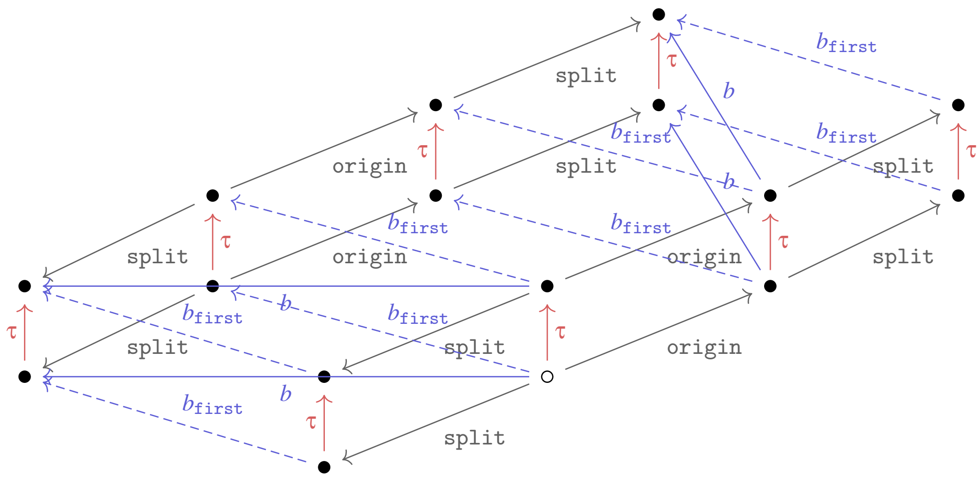
A counterexample to the encoding of the Interrupt Operator being Strongly Bisimilar is with the trivial example

$$P = \tau, Q = b$$

This should yield the following process graph:

$$\begin{tikzcd}
\circ \arrow[r, "\tau"] \arrow[d, "b"] & \bullet \arrow[d, "b"] \\
\bullet & \bullet
\end{tikzcd}$$

However, it yields the following



Which reduces down to:

$$\begin{tikzcd}
\circ \arrow[r, "\tau"] \arrow[d, "b"] & \bullet \arrow[d, "b"] \\
\bullet \arrow[r, "\tau"] & \bullet
\end{tikzcd}$$

A.3.2 Throw Operator

A counterexample to the encoding of the Throw Operator being valid up to Strong Bisimularity is the process

$$P = \tau, Q = b$$

$$
\begin{tikzcd}
\bullet \arrow[r, dashed, "\texttt{split}"] \arrow[d, dashed, "b_{\texttt{split}}"] \arrow[dr, "b"] & \bullet \arrow[r, "q_1"] \arrow[d, dashed, "b_{\texttt{split}}"] & \bullet \arrow[d, dashed, "b_{\texttt{split}}"] \\
\bullet \arrow[r, dashed, "\texttt{split}"] \arrow[d, "\tau"] & \bullet \arrow[r, "q_1"] \arrow[d, "\tau"] & \bullet \arrow[d, "\tau"] \\
\bullet \arrow[r, dashed, "\texttt{split}"] & \bullet \arrow[r, "q_1"] & \bullet
\end{tikzcd}
$$

Which reduces down to

$$
\begin{tikzcd}
\bullet \arrow[dr, "b"] & & \\
& \bullet \arrow[r, "q_1"] \arrow[d, "\tau"] & \bullet \arrow[d, "\tau"] \\
& \bullet \arrow[r, "q_1"] & \bullet
\end{tikzcd}
$$

Appendix B

Extended Definitions

B.1 Extended Definitions

B.1.1 Operational Semantics for ACP^{τ}_{F}

Shown in Figure B.1 is the full set of Operational Semantics for the language ACP^{τ}_{F}.

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

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

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

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

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

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

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

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

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

$$\frac{\langle S_X \mid S \rangle \xrightarrow{\alpha} \surd}{\langle X \mid S \rangle \xrightarrow{\alpha} \surd} \qquad \frac{\langle S_X \mid S \rangle \xrightarrow{\alpha} P'}{\langle X \mid S \rangle \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} \surd}{f(P) \xrightarrow{f(a)} \surd} \qquad \frac{P \xrightarrow{\alpha} P'}{f(P) \xrightarrow{f(a)} f(P')}$$

Table B.1: Extended Structural operational semantics of the language ACP_F^τ. Compared to Table 3.2, this includes additional rules for Successful Termination, $\surd$, which are highlighted in red.