

Expressive specification and verification of choreographies



Open Universiteit



Universiteit
Leiden
The Netherlands

The work in this thesis has been carried out at the Open University of the Netherlands, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). Parts of the research were conducted at the research institute for mathematics and computer science in the Netherlands (CWI) and at the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University.

Typeset using \LaTeX

Cover illustration by Giada Canu, Westend61

Printed by Ridderprint, www.ridderprint.nl

ISBN 978-94-6506-597-7

IPA Dissertation Series 2024-11

Expressive specification and verification of choreographies

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. dr. Th.J. Bastiaens
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op donderdag 12 december 2024 te Heerlen
om 13.30 precies

door
Luc Johan Edixhoven
geboren op 23 juli 1995 te Rennes, Frankrijk

Promotores:

Dr. B.J. Heeren
Prof. dr. M.M. Bonsangue

Open Universiteit
Universiteit Leiden

Copromotor:

Dr. S.T.Q. Jongmans

Open Universiteit

Leden beoordelingscommissie:

Prof. dr. W.J. Fokkink
Prof. dr. F. Montesi
Prof. dr. J.A. Pérez Parra
Prof. dr. T. Vos
Dr. I. Castellani
Dr. T. Kappé

Vrije Universiteit Amsterdam
Syddansk Universitet, Denemarken
Rijksuniversiteit Groningen
Open Universiteit
INRIA, Frankrijk
Open Universiteit

Contents

1	Introduction	1
1.1	Background	1
1.2	This thesis	10
1.2.1	Specification	11
1.2.2	Verification	13
1.3	Publications	18
2	Preliminaries	21
3	Expressive choreography languages	27
3.1	Introduction	27
3.2	Balanced regular languages	30
3.3	Balanced-by-construction regular languages	35
3.4	Balanced ω -regular languages	44
3.5	Balanced-by-construction ω -regular languages	50
3.6	Conclusion	58
4	Branching pomsets	61
4.1	Syntax	61
4.2	Semantics	64
4.3	Comparison with event structures	68
4.3.1	Event structure landscape	68
4.3.2	Comparison – Overview	80
4.3.3	Comparison – Static models	81
4.3.4	Comparison – Dynamic models	84
4.4	Conclusion	86

5	Branching pomsets for choreographies	89
5.1	Introduction	89
5.2	Choreography language definition	90
5.3	BP encoding	94
5.4	Equivalence of BP encoding	97
5.5	Conclusion	100
6	Realisability of branching pomsets	101
6.1	Introduction	101
6.2	Realisability	102
6.3	Well-formedness	106
6.4	Realisability proof	109
6.5	Examples	113
6.6	Related work	114
6.7	Conclusion	116
7	Implementation	119
7.1	Using the B-Pomset Encoder	120
7.2	Realising well-formedness	122
7.3	Performance evaluation	124
7.4	Future work	126
8	Conclusion	127
8.1	Research questions	127
8.2	Future work	128
A	Additional proofs for Chapter 3	131
B	Dynamic causality event structures	139
	References	143
	Author publications	153
	Summary	155
	Samenvatting	157
	Acknowledgements	159
	Curriculum Vitae	163

Introduction

1.1 Background

A day at the seaside

The French children’s book ‘Caroline à la mer’ [Pro65], whose original cover is shown in [Figure 1.1](#), tells of the adventures of a young girl, Caroline, and her friends during a day at the seaside. Depicted on the cover are Caroline herself, along with five of her friends: Youpi (the pup wearing the straw hat), Pouf (the white kitten holding hands — or paws — with Caroline), Noiraud (the black kitten with the bird-shaped swim tube), Boum (the bear cub bending down to pick up something) and Kid (the lion cub in the back). Not pictured are Bobi (a black-and-white pup), Pitou (a panther cub) and Pipo (a sheep dog pup).¹ In the book, amongst other things, the group plays on the beach, goes diving, and rescues Youpi after he gets surprised by the high tide and finds himself stuck on a rock surrounded by water. Not shown in the book, unfortunately, is any use of the large beach ball that Caroline is holding on the cover. In part, this thesis aims to fill this gap.

Imagine that some members of the group are playing with the ball by passing it to one another. A typical element of such a game is to make sure that no-one

¹These are the animals’ original names in French. They may differ depending on your localisation.



Figure 1.1: The cover of our copy of ‘Caroline à la mer’.

feels left out, and thus that everyone receives the ball every now and then (this is known as *fairness*). Generally the group will pay some attention to this and will try to pass the ball to someone if they feel they are not receiving it often enough. Caroline and her friends, however, decide to be more rigorous, and they come up with a straightforward solution: they agree on a fixed order in which

they receive the ball — say, Caroline-Youpi-Pouf-Noiraud-Boum, assuming these are the five playing — and then keep passing the ball in this order until they grow tired of playing.

A day at the seaside, formalised

Formally, Caroline and her friends can be modelled as a *distributed system*. A distributed system consists of a number of *communicating processes* that collaborate to perform a common task. Communication in such a system traditionally consists of passing messages between processes. When modelled as a distributed system, the processes are Caroline and her friends, who communicate by passing the beach ball to one another, and their common task is (presumably) to have fun. The system’s behaviour can also be formalised: for example, the earlier solution in which the group continuously passes the ball in a fixed order can be modelled as the state machine in [Figure 1.2](#).

The state machine depicts the possible locations of the ball as circles (*states*), numbered upwards from 0, and the ways to transition from one state to another by way of the labelled arrows (*transitions*). Initially, the machine starts in state 0, as indicated by the small external arrow; in state 0, Caroline is holding the ball. Caroline can then pass the ball to Youpi, depicted in the state machine by the arrows to states 1 and then 2. The passing of messages in this machine (and the remainder of this thesis) is *asynchronous*, meaning that each communication consists of two distinct parts — a *sending action* and a *receiving action* (or *send resp. receive* for short) — which do not happen simultaneously. The sending part of the communication between Caroline and Youpi is depicted by the arrow from state 0 to state 1, which is labelled with $CY! \text{🏐}$: CY to denote a message from Caroline to Youpi, and $!$ to denote that this is the sending part — and, naturally, 🏐^2 to denote that the message itself consists of a beach ball. Dually, the arrow from state 1 to state 2 is labelled with $CY? \text{🏐}$, where CY still indicates a message from Caroline to Youpi, and where $?$ denotes the receiving part of the communication. Consequently, state 1 represents the situation after Caroline has passed the ball but before Youpi has received it, and state 2 represents Youpi holding the ball. The machine then goes on with Youpi passing the ball to Pouf, Pouf receiving the ball and so forth. A number of intermediate states have been omitted to avoid clutter. The cycle ends with Caroline receiving the ball from Boum, after which the state machine is once again in state 0 (“Caroline has the ball”). Caroline can then again throw the ball to Youpi to initiate another cycle; this can be repeated arbitrarily often.

² 🏐 by AomAm, www.iconfinder.com/aomam.ss, www.iconfinder.com/icons/2138269.

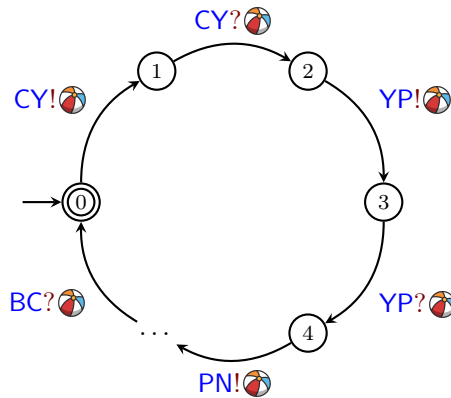


Figure 1.2: A state machine modelling Caroline and her friends passing the ball in a fixed order.

Alternatively, the group could now stop, which is denoted by state 0 being doubly circled: it is a final state, i.e., the state machine may terminate while in this state.

It is worth looking in more detail at asynchronous communication and messages, and introducing the concept of *concurrency*. To illustrate this, we give the group a second beach ball. The group passes this second ball in the same order as the first one, except that it initially starts with Boum. The beach balls are rather large, and Caroline and her friends rather small, so each of them can only hold a single ball at any time. Figure 1.3 shows part of the corresponding state machine. We can then observe several things:

- The two cycles are *concurrent*: they can both make progress independently of the other. From the initial state, 00, in which Caroline and Boum have the balls, either of the two can throw their ball, thus moving to either state 10 or 01.³ Note that there is no state 02, which would correspond to Caroline receiving a ball from Boum before throwing her own, since each player is limited to one ball at a time.
- Since the communication is asynchronous, not only does it consist of two separate parts (send and receive), but the two parts also do not have to happen back-to-back. This is illustrated by the path from state 00 to 01,

³In a truly concurrent system, the two actions would also be able to occur simultaneously, thus moving directly from state 00 to 11. For the sake of simplicity, we do not model this.

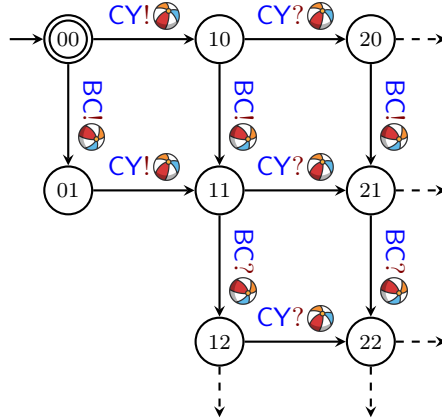


Figure 1.3: Part of a state machine modelling Caroline and her friends passing two balls in a fixed order, with players being able to hold one ball at a time.

11 and then 12: Boum throws a ball to Caroline, after which Caroline first throws her own ball and then receives the one from Boum. From state 12, Caroline could then throw a second ball to Youpi, thus moving to state 13 (not shown). Subsequently, both balls would be in transit from Caroline to Youpi, who can then receive them and pass them on one by one.

- The labels in the state machine do not distinguish between the two balls. In reality, naturally, the two are distinct, but in the model both are represented as . Formally, we say that the players communicate messages of *type* .

The state machines in Figures 1.2 and 1.3 model the behaviour of the group as a whole. We can similarly model the behaviour of the individual participants; for example, Figure 1.4 shows state machines for the behaviour of Caroline and Youpi in Figure 1.2. Caroline distinguishes two states: either she has the ball (state C_0) or she does not (state C_1). Realistically, on the beach Caroline would be able to observe her friends and would know which of them currently has the ball — or between which pair of friends the ball is currently sailing through the air. However, in distributed systems we assume that all the processes can only observe their own, local, behaviour. Caroline's state C_0 thus corresponds to state 0 in Figure 1.2, and state C_1 represents all the others. Initially, Caroline has the ball: she thus starts in state C_0 . When she has the ball, she may pass



Figure 1.4: State machines modelling the local perspectives of two players for the state machine in [Figure 1.2](#).

it to Youpi, after which she is in state C_1 . From state C_1 , she will at some point receive the ball from Boum, after which she returns to state C_0 . The group may stop when Caroline has the ball, so C_0 is also Caroline’s final state. Similarly, Youpi distinguishes two states: either he has the ball (state Y_0) or he does not (state Y_1). State Y_0 corresponds to state 2 in [Figure 1.2](#), while state Y_1 represents all the others. Initially, he does not have the ball, and he thus starts in state Y_1 . He moves to state Y_0 by receiving the ball from Caroline, and then back to Y_1 by passing the ball to Pouf. The group may stop when Youpi does not have the ball (i.e., it may stop when Caroline has it), so Y_1 is Youpi’s final state. All the other participants’ state machines are analogous to Youpi’s. Interestingly, the individual state machines for [Figure 1.3](#) are almost identical to the ones for [Figure 1.2](#): since the friends can only hold one ball, they do not distinguish any more states than before. The only difference is that Boum now starts (and finishes) in a state in which he has a ball.

Having a formal model enables formal reasoning. For example, we can prove that, in each completed sequence of actions of the state machine in [Figure 1.2](#) (i.e., a path following the arrows through the state machine and ending in a final state), each of the friends receives the ball equally often — which is the fairness property they initially set out to achieve. We can also prove that someone (in this case Caroline) is holding the ball when the group stops, and thus that the ball is not forgotten.

Specifying communication protocols as choreographies

The earlier agreed-upon “repeatedly pass the ball in the order Caroline-Youpi-Pouf-Noiraud-Boum”, while intuitive, is also ambiguous. For example, it does not specify whether the friends should complete at least one cycle; the state machine in [Figure 1.2](#) can terminate immediately, before any communication has happened, which corresponds to zero iterations of the cycle. Neither does

the agreement specify whether the game should always end with Caroline, or whether the group may stop at any point; the state machine assumes that they may only stop after a complete cycle, i.e., when Caroline has the ball. Allowing the group to stop at any point can be modelled by making every state a final state — or, preferably, every even state to prevent the ball from being left on the beach. When formally reasoning about the behaviour of systems, it is thus important to have a precise description of their intended behaviour.

A *communication protocol* prescribes the intended communications in a system. There exist many different notations for protocols (or protocol *specifications*), both visual (such as message sequence charts [ITU11] and business process model and notation⁴) and textual (such as multiparty session types [HYC08]); we will focus on the latter. Formally, the repeated cyclic passing of the ball could be specified as follows:

$$(C \rightarrow Y: \text{ball}; Y \rightarrow P: \text{ball}; P \rightarrow N: \text{ball}; N \rightarrow B: \text{ball}; B \rightarrow C: \text{ball})^* \quad (1.1)$$

Here, $C \rightarrow Y: \text{ball}$ denotes a communication (of a ball) from Caroline to Youpi, ‘;’ denotes things happening in sequence (sequential composition), and the closing ‘*’ denotes that everything inside the parentheses should be repeated zero or more times (though only finitely often). This precisely matches the behaviour in Figure 1.2. We make two important observations about this protocol specification:

1. It specifies the behaviour of the entire system, as opposed to that of the individual participants. In other words: it directly specifies the *global perspective* given in Figure 1.2, not the *local perspectives* given in Figure 1.4. We also say that it is a *global specification*, whereas the specifications of the participants are *local specifications*.
2. It specifies behaviour in terms of full communications, as opposed to the individual sending and receiving actions in Figure 1.2.

Specifications with the properties described above are called *choreographies*. Their use is well-established [ITU11; AEY05; HYC08; HYC16; CM13; CM20], and offers several benefits. Designing a system from a global perspective yields a clearer specification and easier reasoning, and allows to directly modify the global specification. The matching local specifications for the processes, or even behaviourally equivalent implementations, can then be automatically derived from the global one through a process called *projection*. In contrast, when designing a system from the local perspectives of the processes, the global

⁴<https://www.omg.org/spec/BPMN>

perspective is derived from the local ones through a process called *composition*. Furthermore, choreographies provide certain safety guarantees by default. Since they specify full communications, it is impossible to specify a *communication error*: a send-receive pair with mismatching messages, or a send or receive without its counterpart.⁵ Choreographies also provide *deadlock-freedom*, a property stating that the system is always able to eventually make progress. Certain choreographies also provide the stronger property of *starvation-freedom*, which deals with the progress of individual processes, rather than the progress of the system as a whole.

Choreography compliance and realisability

Unfortunately, the properties of choreographies do not always carry over to their projections. To illustrate this with an example, consider the simple protocol from before, in (1.1). While safe, it is also repetitive and, arguably, boring.⁶ Caroline and her friends decide to add some variety to it: they add a second (fixed) order and will arbitrarily alternate between the two. This way, they still guarantee fairness but they also prevent every cycle from being the same. Their second possible order swaps the positions of Youpi and Boum with respect to the first possible order, and is otherwise the same. Formally, the new protocol they agree on is as follows:

$$\begin{aligned} & ((C \rightarrow Y: \text{ball}; Y \rightarrow P: \text{ball}; P \rightarrow N: \text{ball}; N \rightarrow B: \text{ball}; B \rightarrow C: \text{ball}) \\ & + (C \rightarrow B: \text{ball}; B \rightarrow P: \text{ball}; P \rightarrow N: \text{ball}; N \rightarrow Y: \text{ball}; Y \rightarrow C: \text{ball}))^* \end{aligned} \quad (1.2)$$

Communications, sequential composition (‘;’) and finite repetition (‘*’) are as before. New is ‘+’, which denotes a (nondeterministic) choice between the two orders. While there are multiple ways to model choices, we choose to leave the manner in which the group decides upon an order unspecified. Note that the ‘*’ now applies to the outer pair of parentheses, which encloses the choice: the choice is repeated zero or more times. Furthermore, there is no mechanism that forces the two orders to be picked equally often; it is, for example, possible for the second to be picked every time. Figure 1.5 shows a state machine modelling this new behaviour. It is analogous to the state machine in Figure 1.2, except that there are two outgoing transitions from state 0, each starting one of the two possible cycles.

⁵In the literature, sends and receives without counterparts are also called *lost messages* and *orphan messages*, respectively.

⁶Some people may argue that playing with a beach ball will be boring regardless of the approach. Those people should be thoroughly ignored.

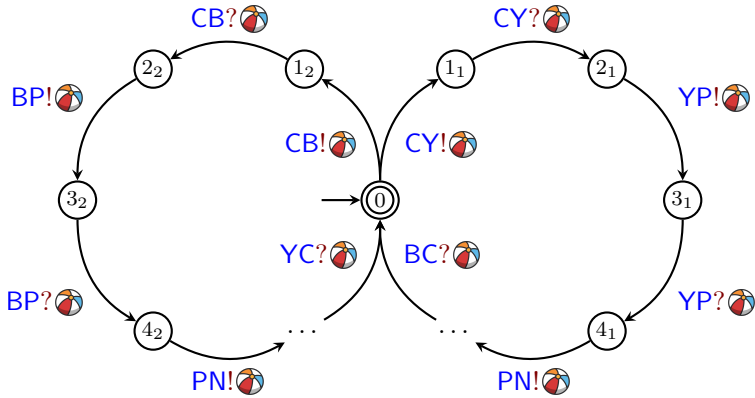


Figure 1.5: A state machine modelling Caroline and her friends passing the ball in one of two fixed orders.

Upon executing this protocol, Caroline and her friends run into a problem. The problem appears when considering the behaviour of Noiraud, modelled in [Figure 1.6a](#). The two outgoing transitions from Noiraud's initial state, N_0 , are both receives of the ball from Pouf. Subsequently, Noiraud should pass the ball to either Boum or Youpi, depending on which of the two originally passed the ball to Pouf. However, Noiraud cannot observe which of the two passed the ball to Pouf and will thus have to pick one of the two without this information; since he cannot distinguish the two cycles based on his local information, he is thus not guaranteed to follow the same cycle as the one picked by Caroline.

To illustrate the problems this can cause, consider the state machines for Noiraud, Youpi and Caroline in [Figure 1.6](#). Suppose that Caroline initially passes the ball to Youpi, thus moving from state C_0 to C_1 — this corresponds to the right cycle in [Figure 1.5](#). Youpi will then receive the ball and pass it to Pouf, thus moving from state Y_0 to Y_1 and back to Y_0 . Pouf will forward the ball to Noiraud. If Noiraud guesses wrong and instead decides to follow the left cycle, then he receives the ball, moving from state N_0 to N_2 , and then moves back to N_0 by throwing the ball to Youpi. From Youpi's point of view (he is in state Y_0), he is able to receive a ball from Noiraud: this could be the second round. Youpi thus receives the ball and throws it to Caroline, thus moving from state Y_0 to state Y_2 and back to Y_0 . Meanwhile, Caroline is still in state C_1 , expecting a ball from Boum. In state C_1 , she cannot receive a ball from Youpi. At this point, since Caroline cannot receive the ball and no other

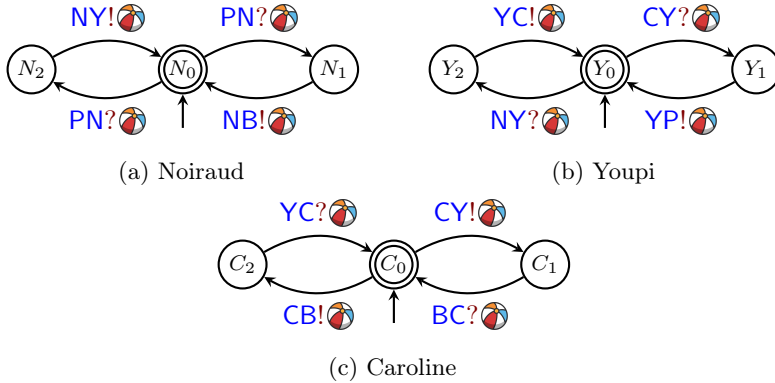


Figure 1.6: State machines modelling the local perspectives of three players for the state machine in [Figure 1.5](#).

member of the group can perform any action either, they are stuck: the system is *deadlocked*.

Recall from earlier that choreographies guarantee an absence of deadlocks. However, the system implementing the choreography in (1.2) still resulted in one. The problem here is that the prescribed behaviour of the choreography does not match the behaviour of its implementation: the system is not *compliant* with the choreography. Relatedly, a choreography is *realisable* if there exists at least one compliant implementation of it, i.e., an equivalent composition of local specifications — otherwise, it is *unrealisable*. While it is thus impossible to *specify* communication errors or deadlocks in a choreography, these guarantees are only meaningful for a concrete system if the system is actually compliant, and if the choreography is realisable in the first place. This thesis focuses on the latter problem, realisability.

We refer the interested reader to Fabrizio Montesi’s recent book ‘Introduction to Choreographies’ [[Mon23](#)] for further reading on choreographies.

1.2 This thesis

The contributions of this thesis to the study of choreographies can be divided into the two parts mentioned in its title: *specification* and *verification*.

1.2.1 Specification

The first part of this thesis studies the specification of choreographies, or, more specifically, *choreography specification languages* — *choreography languages* for short. A choreography language consists of communications (such as $C \rightarrow Y: \text{ball}$) and a number of operators (such as ‘;’, ‘*’ and ‘+’) to combine the communications into the desired patterns. This makes it possible, for example, to specify the choreographies in (1.1) and (1.2), which specify the patterns in Figures 1.2 and 1.5, respectively. The operators in the language determine the patterns that can be specified in it. Informally, we refer to the possible patterns that can be specified in a given choreography language as that language’s *expressiveness*.

The expressiveness of existing choreography languages is typically limited: for each such language, there exist sensible patterns of interaction which cannot be specified in it. In part, this is because of choreography compliance and realisability. Realisability is discussed in more detail in Section 1.2.2; for now it suffices to say that the problem is difficult and that the expressiveness of many languages is syntactically restricted to a subset for which realisability can be more easily established. Another factor in the limited expressiveness of choreography languages is the syntactic constraint of using full communications in expressions. It is well known that all (finite) state machines can be represented using expressions with the operators ‘;’, ‘*’ and ‘+’ [Kle56], if the specification language contains the individual send and receive actions ($CY! \text{ball}$ and $CY? \text{ball}$, respectively). However, it is then also possible to specify communication errors and deadlocks — for example, by specifying $CY? \text{ball}$, a singleton receive without a matching send. Choreography languages rule out communication errors by only allowing full communications. In turn, this also rules out certain communication-error-free (and thus desirable) patterns, which then require the introduction of new operators to express.

To illustrate this, the state machine in Figure 1.7 models an exchange between Caroline and Youpi. First they both throw a ball to the other, in any order, and then they both receive them, again in any order. This kind of pattern occurs in situations where a participant should first send their own message(s) before receiving any, which is important in, for example, distributed games (such as rock-paper-scissors) or a distributed vote. While the behaviour modelled by the state machine does not contain communication errors and is realisable, it is also impossible to express as a choreography when using only ‘;’, ‘*’ and ‘+’. At the very least, we would need the two basic communications, $C \rightarrow Y: \text{ball}$ and $Y \rightarrow C: \text{ball}$. Combining the two with sequential composition (‘;’), however, forces one of them to occur before the other, as seen in Figure 1.8a. Specifications for

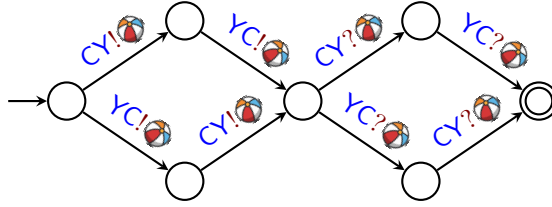


Figure 1.7: A state machine modelling Caroline and Youpi exchanging balls, with neither having both simultaneously.

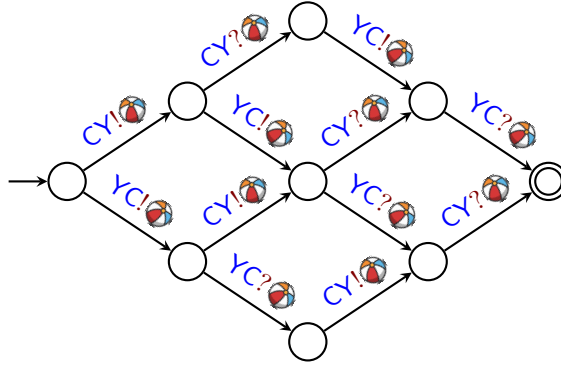
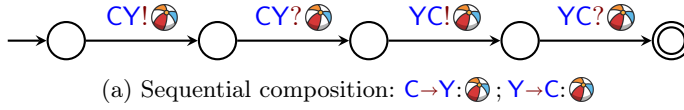


Figure 1.8: State machines modelling the sequential and parallel composition, respectively, of two communications.

concurrent systems often use parallel composition (\parallel), which allows things to happen in any order. The result, shown in Figure 1.8b, resembles Figure 1.7 but includes two additional states, corresponding to the cases where either Caroline or Youpi receives the other's ball before throwing their own. Specifying the behaviour in Figure 1.7 thus requires a different operator, or different operators.

We study the theoretical limits of the expressiveness of choreography languages. Choreographies specify communication-error-free behaviour. The dual question

then remains: how much communication-error-free behaviour can be expressed in choreography languages? Behaviour such as that in [Figure 1.7](#) requires more sophisticated operators, and it might be possible that it is simply unavoidable to lose some expressiveness when using a choreography language. Specifically, we focus on behaviour expressed as a finite state machine, as seen in, e.g., [Figure 1.7](#). Those familiar with formal language theory will know that finite state machines correspond to the class of *regular languages* — or, in our case, *regular protocols*. While there exist other types of state machines, corresponding to other classes of formal languages, we are not aware of existing choreography languages that specify non-regular protocols⁷, and the potential expressiveness of choreography languages is unknown both in general and for regular protocols in particular. In summary, we can thus ask the following question:

Question 1. Does there exist a choreography language capable of specifying all communication-error-free regular communication protocols?

[Chapter 3](#) answers this question positively: we describe a family of operators, called *shuffle on trajectories*, introduced in 1998 by Alexandru Mateescu, Grzegorz Rozenberg and Arto Salomaa [[MRS98](#)], and we show how it can be used in a choreography language to express any (communication-error-free) regular protocol. In other words: we show that it is possible to syntactically enforce an absence of communication errors without losing any expressiveness. Furthermore, we show the same results for the class of ω -regular protocols, which describe infinitely long sequences of behaviour.

1.2.2 Verification

The remainder of this thesis studies the verification of properties of choreographies, in particular realisability. Specifically, we introduce a new model for the behaviour of choreographies, and we use it to study realisability.

Behavioural models

In the past five to ten years, there has been growing interest in behavioural models for choreographies, both for the purpose of analysis [[GT19](#); [SY19](#); [JY20](#)] and for code generation [[HY16](#); [Kou+18](#); [YZF21](#); [Bar+23](#); [FJ23](#)]. Generally speaking, the larger the model is, the longer the analysis takes. Conversely, more compact models may lead to more efficient verification.

⁷Recall that we consider choreography specification languages. This statement does not apply to, e.g., choreographic programming languages.

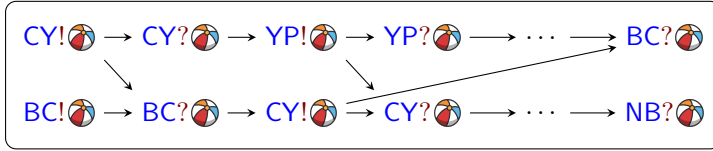


Figure 1.9: A pomset modelling Caroline and her friends passing two balls in a fixed order, with players being able to hold at most one ball at a time.

In contrast, a pomset only represents a single (concurrent) scenario, i.e., a fixed set of events that may occur in a number of orders. Representing a choice between possible events requires a *set* of pomsets: one for every possible combination of branches. For example, one iteration of the protocol in (1.2) would require two pomsets, one for each of the two possible orders. To make matters worse, two iterations would require $2 \times 2 = 4$ pomsets, since there are two possibilities for each of the two iterations. In general, n iterations with k possible orders each require k^n pomsets, which, while a different kind of explosion, is an explosion nonetheless.

Consequently, protocols containing combinations of concurrency and choice will lead to an explosion in the size of both models. This happens if, for example, we would add additional balls to (1.2). This spurs the following question:

Question 2. Can we design a model that compactly represents combinations of concurrency and choice in choreographies?

We answer this question positively by extending pomsets with a choice mechanism, by means of a branching structure. We present and explore the resulting model, *branching pomsets* (*BPs* for short), in Chapter 4. Figure 1.10 shows one iteration of a protocol with two possible cycles and two balls, modelled as a BP. Each white box (except for the outermost one) represents one branch of a choice, while the choice itself is represented by the enclosing blue box. As such, each choice box represents the movements of (one iteration of) one ball, with the arrows between the boxes representing the additional dependencies to ensure that players do not receive a ball while already holding one. Choices can be resolved by dropping one their branches (and removing all the corresponding arrows), after which the events in the remaining branch can occur as in traditional pomsets. This is illustrated in Figure 1.11, in which the upper choice has been resolved to its lower branch. In this model, each additional ball would add one choice box, and each additional possible order would add one branch to each choice.

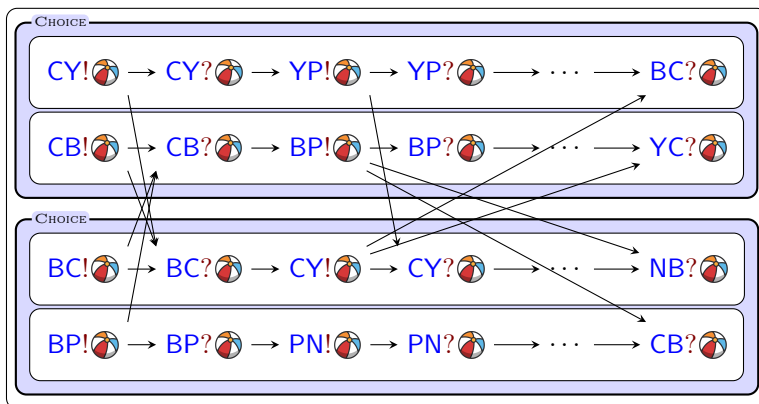


Figure 1.10: A branching pomset modelling Caroline and her friends passing two balls in two possible orders, in which no player holds two balls simultaneously.

We give a thorough comparison between BPs and several existing classes of *event structures* [NPW81], which also generalise pomsets with a choice mechanism.⁸ A major difference is that the branching structure of BPs is hierarchical, resembling choices in expressions in choreography languages. In contrast, most classes of event structures model choices through a conflict relation, which prohibits certain pairs of events of occurring in the same computation. BPs define an interesting new class of behaviour. Their expressiveness turns out to be incomparable with most existing classes of event structures, and they are only contained in the more expressive classes of dynamic event structures.

In [Chapter 5](#) we define a simple choreography language and show how to encode its behaviour in the BP model.

Realisability

The realisability problem has been well-studied in the last two decades in various settings. Notably, Rajeev Alur, Kousha Etessami and Mihalis Yannakakis show that, for bounded MSC-graphs, which define a subset of regular languages, the problem is undecidable for one definition of realisability and PSPACE-hard and in EXPSPACE for another [AEY05].⁹ Markus Lohrey [Loh02] improves the lat-

⁸More accurately, event structures can be seen as a generalisation of posets, and labelled event structures can be seen as a generalisation of pomsets.

⁹There exist multiple definitions of realisability, depending on the type of equivalence used and, for example, depending on whether or not deadlock-freedom is included.

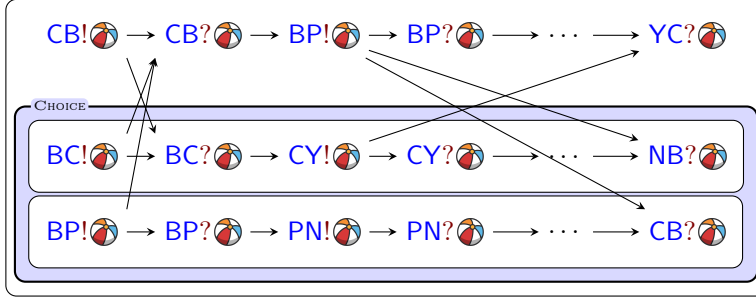


Figure 1.11: The branching pomset in Figure 1.10 after its upper choice is resolved to its lower branch.

ter bound by establishing EXPSPACE-completeness. While there are differences between our settings, this shows that the problem is, in general, complex.

Of particular interest to this thesis is the approach adopted by Kohei Honda, Nobuko Yoshida and Marco Carbone in multiparty session types (MPST) [HYC08]. Their analysis does not use a behavioural model; instead, their choreography language and projection operator (implicitly) define well-formedness conditions directly on the specifications. A certain amount of structure is enforced by the language itself, in particular regarding the specification of choices. A specification in MPST is then well-formed if it can be projected upon each participant. These structural conditions are easy to verify and, in the case of MPST, are sufficient to prove realisability. However, they are also conservative: all well-formed specifications are realisable, but not all realisable specifications are necessarily well-formed. Correct specifications may thus be unnecessarily rejected. There has since been plenty of research aimed at relaxing these conditions to reject fewer correct specifications, while still disallowing any incorrect ones; this is briefly discussed in Section 6.6. However, it may also be possible to apply the same approach to another model with sufficient structure, such as BPs:

Question 3. Can we define well-formedness conditions on branching pomsets such that they are both easily verifiable and sufficient to prove realisability?

We present these conditions in Chapter 6; inspired by the well-formedness conditions of MPST, they are defined directly on BPs. We formally prove that the choreography represented by any well-formed BP is realisable. Naturally, like in MPST, the conditions are conservative. Nevertheless, we believe this

result is promising. Since BPs are not tied to the syntax of any particular choreography language, any analysis on them — including realisability through well-formedness — can be used for any choreography language, provided there is an encoding from it into BPs: in the words of Guanciale and Tuosto, it is syntax-oblivious. For example, since we have provided an encoding of the choreography language in [Chapter 5](#) into BPs, we can automatically apply these checks to any choreography expressed in it. For the same reason, it may also be possible to define well-formedness conditions more generally on BPs than on any specific syntax, such as the choreography language of MPST. Finally, we discuss some ideas for (correctness-preserving) relaxations of our well-formedness conditions.

[Chapter 7](#) describes an implementation of the work presented in [Chapters 4](#) to [6](#). It explains how to use a companion tool, developed to simulate BPs and analyse their applications to choreographies. In particular, the tool allows a user to (a) construct BPs, either by specifying them directly or by specifying a choreography, (b) explore the possible executions of BPs step-by-step, and (c) automatically check for the presented well-formedness conditions. We also discuss the implementation of these well-formedness conditions, and show that its complexity is much less than that of a full realisability analysis. The tool is available as a live snapshot at <https://lmf.di.uminho.pt/b-pomset/>, and is archived as part of an artefact in the Zenodo repository at <https://doi.org/10.5281/zenodo.7888538>. Many of the examples in [Chapters 4](#) to [6](#) are predefined in the tool, as well as a number that do not feature in this thesis; the digital version of this thesis provides hyperlinks that open the tool with the specific example, e.g., R_c [↗](#) ([Figure 5.3](#), on page 95).

1.3 Publications

The majority of this thesis consists of published material for which I was the lead author, either in conference or workshop proceedings or in journals. The start of each chapter mentions the relevant publications, as well as any (major) additions. The following is an overview of the papers upon which this thesis is based:

- [\[DLT21\]](#) Luc Edixhoven and Sung-Shik Jongmans. ‘Balanced-By-Construction Regular and ω -Regular Languages’. In: *Developments in Language Theory - 25th International Conference, DLT 2021, Porto, Portugal, August 16-20, 2021, Proceedings*. Ed. by Nelma Moreira and Rogério Reis. Vol. 12811. Lecture Notes in Computer Science. Springer, 2021, pp. 130–142. DOI: [10.1007/978-3-030-81508-0_11](https://doi.org/10.1007/978-3-030-81508-0_11)

- [IJFCS23] Luc Edixhoven and Sung-Shik Jongmans. ‘Balanced-by-Construction Regular and ω -Regular Languages’. In: *Int. J. Found. Comput. Sci.* 34.2&3 (2023), pp. 117–144. DOI: [10.1142/S0129054122440026](https://doi.org/10.1142/S0129054122440026)
- [ICE22] Luc Edixhoven, Sung-Shik Jongmans, José Proença and Guillermina Cledou. ‘Branching Pomsets for Choreographies’. In: *Proceedings 15th Interaction and Concurrency Experience, ICE 2022, Lucca, Italy, 17th June 2022*. Ed. by Clément Aubert, Cinzia Di Giusto, Larisa Safina and Alceste Scalas. Vol. 365. EPTCS. 2022, pp. 37–52. DOI: [10.4204/EPTCS.365.3](https://doi.org/10.4204/EPTCS.365.3)
- [FACS22] Luc Edixhoven and Sung-Shik Jongmans. ‘Realisability of Branching Pomsets’. In: *Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings*. Ed. by Silvia Lizeth Tapia Tarifa and José Proença. Vol. 13712. Lecture Notes in Computer Science. Springer, 2022, pp. 185–204. DOI: [10.1007/978-3-031-20872-0_11](https://doi.org/10.1007/978-3-031-20872-0_11)
- [JLAMP24] Luc Edixhoven, Sung-Shik Jongmans, José Proença and Ilaria Castellani. ‘Branching pomsets: Design, expressiveness and applications to choreographies’. In: *J. Log. Algebraic Methods Program.* 136 (2024), p. 100919. DOI: [10.1016/J.JLAMP.2023.100919](https://doi.org/10.1016/J.JLAMP.2023.100919)

The conference paper [DLT21] and its extended journal version, [IJFCS23], form the basis for [Chapter 3](#). The workshop paper [ICE22] and the conference paper [FACS22] are both included in the later journal paper [JLAMP24], which forms the basis for [Chapters 4 to 7](#). Specifically, [Chapters 4 and 5](#) are based on the material from [ICE22] and additional material from [JLAMP24], [Chapter 6](#) is based on material from [FACS22], and [Chapter 7](#) is based on new material from [JLAMP24].

The following is an overview of further papers to which I have contributed during my PhD, but which are not included in this thesis:

- [ECOOP22] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans and José Proença. ‘API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3’. In: *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*. Ed. by Karim Ali and Jan Vitek. Vol. 222. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 27:1–27:28. DOI: [10.4230/LIPICS.ECOOP.2022.27](https://doi.org/10.4230/LIPICS.ECOOP.2022.27)

- [COORD23] José Proença and Luc Edixhoven. ‘Caos: A Reusable Scala Web Animator of Operational Semantics’. In: *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*. Ed. by Sung-Shik Jongmans and Antónia Lopes. Vol. 13908. Lecture Notes in Computer Science. Springer, 2023, pp. 163–171. DOI: [10.1007/978-3-031-35361-1_9](https://doi.org/10.1007/978-3-031-35361-1_9)
- [iFM23] Luc Edixhoven. ‘Shuffling Posets on Trajectories’. In: *iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands, November 13-15, 2023, Proceedings*. Ed. by Paula Herber and Anton Wijs. Vol. 14300. Lecture Notes in Computer Science. Springer, 2023, pp. 384–390. DOI: [10.1007/978-3-031-47705-8_21](https://doi.org/10.1007/978-3-031-47705-8_21)

The conference paper [ECCOOP22] presents a way to leverage new functionality in the Scala programming language to generate compact code for processes in choreographies. This application of choreographies is briefly discussed in [Chapter 8](#). The conference paper [COORD23] presents a programming framework in Scala, which is used by the tool described in [Chapter 7](#), aimed at visualising, animating and analysing the behaviour of abstract models to support research and teaching endeavours. I was a co-author but not a main contributor on these two papers. The conference paper [iFM23], of which I am the only author, presents a (very) first step towards combining the main results of this thesis: it explores how the shuffle on trajectories operator, shown in [Chapter 3](#) to be potentially useful in choreography languages, can be applied to posets, a simpler version of the BP model presented in [Chapter 4](#). This is briefly discussed in [Chapter 8](#).

Preliminaries

In this chapter we introduce common notation and terminology used throughout the thesis.

Sets, multisets and numbers

We write \mathbb{N} for the set of natural numbers (nonnegative integers), $\{0, 1, 2, \dots\}$. We write \mathbb{N}^+ for the set of positive integers, $\{1, 2, \dots\}$. We write \mathbb{Z} for the set of integers, $\{\dots, -1, 0, 1, \dots\}$. We write \emptyset for the empty set. We write $\mathcal{P}(X)$ for the power set of a set X , and $X \setminus Y$ for set difference. If X is a multiset, we write $e^{(k)} \in X$ to indicate that e occurs in X with multiplicity $k \in \mathbb{N}$, and $\text{mult}(e, X)$ to denote the multiplicity of e in X . Note that k may be equal to 0, in which case $e^{(0)} \in X$ means that e is not an element of X . We write $|X|$ for the cardinality of a set or multiset X . We write \aleph_0 for countable infinity, i.e., $|\mathbb{N}|$.

Alphabets, words and languages

Let Σ be a finite set of symbols, or alphabet. We will define precise alphabets in the relevant chapters. A word over Σ is a finite or infinite sequence of symbols from Σ . We write ε for the empty word. We write Σ^* for the set of finite words over Σ , Σ^ω for the set of (strictly) infinite words over Σ , and Σ^∞ for $\Sigma^* \cup \Sigma^\omega$.

A language L over Σ is a set of words over Σ . A language is a subset of either Σ^* or Σ^ω , but we sometimes write $L \subseteq \Sigma^\infty$ as a shorthand to cover both cases.

Concatenation of two words is written $v \cdot w$, though we often omit the \cdot and simply write vw . Similarly, we write the concatenation of languages as either $L_1 \cdot L_2$ or simply L_1L_2 . For any concatenation of words vw we assume that $v \in \Sigma^*$, and for any concatenation of languages L_1L_2 we assume that $L_1 \subseteq \Sigma^*$.

We write $w(i)$ to refer to the symbol at position i in $w \in \Sigma^\infty$, with $1 \leq i \leq |w|$ if $w \in \Sigma^*$ and $i \in \mathbb{N}^+$ if $w \in \Sigma^\omega$. We write $w(i, j)$ for the substring of $w \in \Sigma^\infty$ beginning at position i and ending at position j (inclusive). For $v \in \Sigma^*$ and $w \in \Sigma^\infty$, v is a prefix of w , denoted $v \preceq w$, if there exists $v' \in \Sigma^\infty$ such that $vv' = w$; v is a strict prefix of w , denoted $v \prec w$, if $v \preceq w$ and $v \neq w$. For $w \in \Sigma^\infty$ and $\sigma \in \Sigma$, we write $|w|, |w|_\sigma \in \mathbb{N} \cup \{\aleph_0\}$ respectively for the length of w and for the number of occurrences of symbol σ in w .

Regular and ω -regular expressions

We write \mathbb{E} and Ω respectively for the sets of all regular and ω -regular expressions (over Σ), as defined in [Definitions 2.1](#) and [2.3](#), respectively. We use \emptyset for the term representing the empty language. We write $L(e)$ for the language of an expression $e \in \mathbb{E} \cup \Omega$, defined in [Definitions 2.2](#) and [2.4](#). For $e_1, e_2 \in \mathbb{E} \cup \Omega$, we write $e_1 \equiv e_2$ for $L(e_1) = L(e_2)$.

Definition 2.1 (regular expressions).

$$\frac{}{\emptyset \in \mathbb{E}} \quad \frac{}{\varepsilon \in \mathbb{E}} \quad \frac{\sigma \in \Sigma}{\sigma \in \mathbb{E}} \quad \frac{e_1, e_2 \in \mathbb{E}}{e_1 \cdot e_2 \in \mathbb{E}} \quad \frac{e_1, e_2 \in \mathbb{E}}{e_1 + e_2 \in \mathbb{E}} \quad \frac{e \in \mathbb{E}}{e^* \in \mathbb{E}}$$

Definition 2.2 (language of a regular expression).

$$\begin{aligned} L(\emptyset) &= \emptyset & L(e_1 \cdot e_2) &= L(e_1) \cdot L(e_2) \\ L(\varepsilon) &= \{\varepsilon\} & L(e_1 + e_2) &= L(e_1) \cup L(e_2) \\ L(\sigma) &= \{\sigma\} & L(e^*) &= \{w_1 \dots w_n \mid n \in \mathbb{N}, \forall i \in [1, n] : w_i \in L(e)\} \end{aligned}$$

Definition 2.3 (ω -regular expressions).

$$\frac{}{\emptyset \in \Omega} \quad \frac{e \in \mathbb{E} \quad \varepsilon \notin L(e)}{e^\omega \in \Omega} \quad \frac{e_1 \in \mathbb{E} \quad e_2 \in \Omega}{e_1 \cdot e_2 \in \Omega} \quad \frac{e_1, e_2 \in \Omega}{e_1 + e_2 \in \Omega}$$

Definition 2.4 (language of an ω -regular expression).

$$\begin{aligned} L(\emptyset) &= \emptyset & L(e^\omega) &= \{w_1 w_2 \dots \mid \forall i \in \mathbb{N}^+ : w_i \in L(e)\} \\ L(e_1 \cdot e_2) &= L(e_1) \cdot L(e_2) & L(e_1 + e_2) &= L(e_1) \cup L(e_2) \end{aligned}$$

Automata

Unless otherwise specified, we write “finite automaton” to refer to a nondeterministic finite automaton with empty (i.e., ε -labelled) transitions, as defined in [Definition 2.5](#).

Definition 2.5 (finite automaton). A finite automaton is a five-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an alphabet, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is a transition relation, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states.

If $(p, \sigma, q) \in \delta$, we also write $p \xrightarrow{\sigma} q$. Similarly, we write $p \xrightarrow{w}^* q$ if $w = \sigma_1 \sigma_2 \dots \sigma_n$ and $p \xrightarrow{\sigma_1} p' \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_n} q$. If $q_0 \xrightarrow{w}^* q$, then we also refer to the sequence of states visited as a run of the automaton; if $q \in F$, then we say that it is an accepting run. We sometimes write $p \not\xrightarrow{w} q$ if $(p, \sigma, q) \notin \delta$, and $p \not\xrightarrow{w}^* q$ if there exists no q such that $(p, \sigma, q) \in \delta$.

ω -Automata are automata accepting languages of infinite words. As with finite automata, there are multiple classes of ω -automata. Unless otherwise specified, we will use nondeterministic Muller automata [[Mul63](#)], as defined in [Definition 2.6](#). In the remainder of this thesis we will omit ‘nondeterministic’ and simply write Muller automaton. A Muller automaton differs from a finite automaton, as defined above, only in its acceptance criterion: instead of a set of final states, a Muller automaton has a set of sets of states F , and it accepts exactly those runs in which the set of states that are visited infinitely often is a member of F .

Definition 2.6 (Muller automaton). A Muller automaton is a five-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q, Σ, δ and q_0 are as for finite automata, and $F \subseteq \mathcal{P}(Q)$ is a set of accepting sets of states.

The language of a finite or Muller automaton consists of all words over its alphabet that have at least one accepting run. Note that these words are always finite for finite automata and infinite for Muller automata. Finite automata characterise regular languages, while Muller automata characterise ω -regular languages [McN66].

As for regular and ω -regular expressions, we write $L(M)$ for the language of an automaton or ω -automaton M , and we write $M_1 \equiv M_2$ for $L(M_1) = L(M_2)$. Furthermore, we overload the notation to also compare expressions and automata, e.g., we write $e \equiv M$ for $L(e) = L(M)$.

Labelled transition systems and bisimulation

We use labelled transition systems (LTSs) to define the semantics of branching pomsets in [Chapter 4](#) and of the choreography language presented in [Chapter 5](#). For the purpose of this thesis, LTSs only differ from finite automata as defined in [Definition 2.5](#) in that their set of states Q may be infinitely large. Furthermore, we write $p \downarrow$ to denote that a transition system may terminate in state p , rather than writing $p \in F$.

Typically, we will compare LTSs using bisimulation equivalence [Mil80]. Specifically, since we do not model empty transitions, we use strong bisimulation. Two LTSs are language equivalent (or trace equivalent) if their languages are the same, i.e., if they accept the same set of words (or traces), regardless of the way these words are obtained. On the other hand, two LTSs are bisimilar if their internal branching behaviour is also the same. This is a stronger notion of equivalence than language equivalence: if two LTSs are bisimilar then they are also language equivalent, but the inverse is not necessarily true. Formally, two transition LTSs A_1, A_2 are bisimilar, written $A_1 \sim A_2$, if there exists a bisimulation relation \mathcal{R} between the states of A_1 and A_2 which relates their initial states. The relation \mathcal{R} is a bisimulation relation if, for every pair of states $(p, q) \in \mathcal{R}$:

- If $p \xrightarrow{\sigma} p'$ then $q \xrightarrow{\sigma} q'$ and $(p', q') \in \mathcal{R}$ for some q' , and vice-versa.
- If $p \downarrow$ then $q \downarrow$, and vice-versa.

In other words: if one of the two can perform a step, then the other can perform a matching step such that the resulting states are again related. While the two LTSs may not be identical, they are behaviourally indistinguishable to an external observer.

Posets and pomsets

A partial order is a reflexive, antisymmetric and transitive relation, typically written \leq . A partially ordered set (X, \leq) , or poset for short, is a set X endowed with a partial order \leq on its elements. A labelled poset (X, \leq, λ) , or lposet for short, is a poset with the addition of a labelling function λ from its elements to a set of labels. Note that, while the elements of the set must be unique, this is not required of their labels. Two lposets are isomorphic if one can be obtained from the other by renaming its elements and preserving the partial order and labelling function accordingly. A partially ordered multiset, or pomset for short, is an isomorphism class of lposets [Pra86]. Pomsets can thus be used instead of lposets to abstract away from the names of their elements. When drawing pomsets and branching pomsets, such as in Figures 1.9 and 1.10, respectively, we draw the labels instead of the elements themselves.

Expressive choreography languages

This chapter is based on our DLT 2021 paper [DLT21] and its extended journal version [IJFCS23]. It answers [Question 1](#). As in the original papers, most proofs have been omitted in favour of proof sketches or highlights. Two new proofs, for [Theorems 3.2](#) and [3.15](#), can be found in [Appendix A](#). Other omitted proofs can be found in the technical report of our DLT 2021 paper [DLT21 TR].

3.1 Introduction

In this chapter we study the expressiveness of choreographies from a formal languages perspective. Choreographies are reminiscent of the Dyck language, or the language of balanced parentheses, in formal language theory. Dyck words, i.e., the words in the Dyck language, consist of a number of opening and closing parentheses¹⁰, in such a way that (1) there are equally many opening and closing parentheses, and (2) when reading the word from left to right, one must always have seen at least as many opening as closing parentheses. This is visualised for the Dyck word $[[[]][[[]]][]$ in [Figure 3.1](#), where the vertical axis shows the number of unmatched opening parentheses up to that point. Note that (1) the plot (starts and) ends on 0, and (2) it never drops below 0.

Analogously, when observing the passing of messages between two processes

¹⁰Note that we use square brackets $[]$ throughout the chapter for typographical reasons, but we consistently refer to these as parentheses.

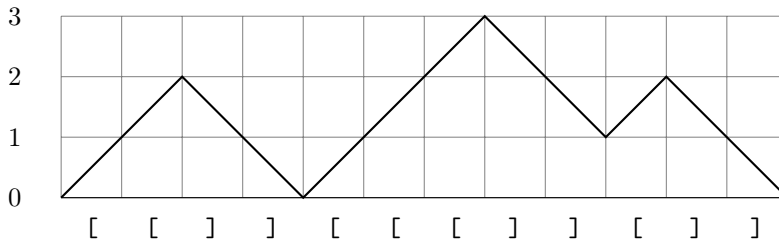


Figure 3.1: Visualisation of the Dyck word $[()][[]][()]$, with the vertical axis showing the number of unmatched opening parentheses up to that point.

on a communication channel, (1) there should be equally many send and receive events of messages on the channel, and (2) when observing the events chronologically, one must always have seen at least as many send as receive events, i.e., messages must be sent before they are received. Therefore, in a sense, the allowed sequences of events on communication channels can be said to be Dyck words. Generalising communications to multiple channels is thus analogous to generalising the Dyck language to multiple types of parentheses.

We briefly discuss a number of existing generalisations. The typical generalisation of the Dyck language to multiple types of parentheses, Paren_n , is central in characterising the class of context-free languages, as shown by the Chomsky-Schützenberger theorem [CS59]. Many other generalisations have been studied over the years. The *unrestricted* Dyck language requires only that the number of left parentheses equals the number of right parentheses, with no restriction on their ordering. Prodingler [Pro79] generalises this unrestricted Dyck language by considering factorisations with arbitrary words instead of with single parentheses. Labelle and Yeh [LY90] allow a larger alphabet where every alphabet symbol is associated with some rational number. A word is then balanced if the value of its letters sums to 0 and if it has no prefix with a negative value. The Dyck language is then the special case with a binary alphabet, where the opening parenthesis has value 1 and the closing parenthesis value -1 . This generalisation is further studied by Duchon [Duc00]. Moortgat [Moo14] also considers a larger alphabet, in which the symbols are ordered $(\sigma_1, \sigma_2, \dots)$. A word is balanced if every prefix of it contains symbol σ_i at least as many times as symbol σ_{i+1} . Finally, Liebehenschel [Lie03] considers a generalisation with multiple types of parentheses where the parentheses are not paired by type but by some similarity relation. Depending on this relation some type of left parenthesis may match multiple types of right parentheses and vice-versa.

However, none of these generalisations capture the properties of choreographies; they all allow either too little or too much. We thus consider our own generalisation with multiple types of parentheses. The parentheses are paired by type and for each type a balanced word must contain the same number of opening and closing parentheses, with no prefix containing more closing than opening parentheses (of that type). The difference with Paren_n is that, while Paren_n requires parentheses of different types to be properly nested, we impose no such restriction: parentheses of different types may freely commute. For example, Paren_n allows $[_1[_2]_2]_1$ but rejects $[_1[_2]_1]_2$; we consider both to be balanced. This notion of balancedness corresponds with the idea that, when you associate each type of parenthesis with a different communication channel, the events on each channel are neatly ordered but there is no ordering between channels. Balancedness then characterises precisely all sequences of communication with no lost or orphan messages. In turn, languages consisting only of balanced words, which we call balanced languages, correspond to choreographies.

We can thus study the expressiveness of choreography languages by studying balanced languages. Specifically, we are then interested in specifying languages that are balanced by construction. More precisely, we aim to answer the question: can we define balanced atoms and a set of balancedness-preserving operators with which one can express all balanced languages?

We answer this question positively for the classes of regular and ω -regular languages. In [Section 3.2](#) we show how balancedness of regular languages corresponds to syntactic properties of finite automata and regular expressions. In [Section 3.3](#) we show that, by using a parameterised shuffle operator, we can define an infinite grammar of balanced-by-construction expressions with which one can express all balanced regular languages. In [Section 3.4](#) we extend balancedness to ω -regular languages and syntactic properties of ω -automata and ω -regular expressions. In [Section 3.5](#) we define a grammar for balanced ω -regular languages. We end the chapter in [Section 3.6](#) with concluding remarks and a brief discussion on future work.

Notation

Throughout this chapter, let Σ_n be the alphabet $\{[_1,]_1, \dots, [_n,]_n\}$. Its size is typically either clear from the context or irrelevant; in both cases we omit the subscript and simply write Σ .

3.2 Balanced regular languages

In this section, we formally define our notion of balancedness and characterize balanced regular languages in terms of finite automata and regular expressions.

Balancedness

A word $w \in \Sigma^*$ is *i-balanced* if $|w|_{\lceil_i} = |w|_{\rfloor_i}$ and if, for all prefixes v of w , $|v|_{\lceil_i} \geq |v|_{\rfloor_i}$. It is *balanced* if it is *i-balanced* for all i . We extend this terminology to languages, automata and expressions in the expected way: a language is (*i*-)balanced if all of its words are; an automaton or expression is (*i*-)balanced if its language is.

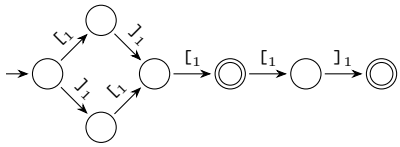
Example 3.1. Figure 3.2 shows examples of balanced and unbalanced languages. The finite automata and regular expressions in the figure can be ignored for now; we discuss them shortly.

The languages in Figures 3.2b, 3.2d and 3.2e are balanced because, informally, in every word of those languages, every \rfloor_i is preceded by a corresponding \lceil_i for every $i \in \{1, 2, 3\}$. In contrast, the language in Figure 3.2a is unbalanced because each of its words w violates requirement $|w|_{\lceil_1} = |w|_{\rfloor_1}$, while the language in Figure 3.2c is unbalanced because word $\rfloor_1 \lceil_1$ violates requirement $|v|_{\lceil_1} \geq |v|_{\rfloor_1}$ for prefix $v = \rfloor_1$.

Finite automata

For a (nondeterministic) finite automaton to be balanced, all words leading to a final state must be balanced: for every i , they must contain equally many opening and closing i -parentheses and the number of closing i -parentheses may never exceed the number of opening i -parentheses in any prefix. It follows that, for any state with a \rfloor_i -transition to a final state, all words leading to this state must contain exactly one unmatched opening i -parenthesis — along with the previous condition on prefixes. Following this line of thought, a similar condition must hold for every state in the automaton: for every state and for every i , there must exist some $n \in \mathbb{N}$ such that $|w|_{\lceil_i} - |w|_{\rfloor_i} = n$ for every word w leading to that state. We will call this n a state's *i-balance* (i.e., the number of unmatched opening i -parentheses), denoted $\nabla(q, i)$ for a state q . Additionally, the *i*-balances of the automaton's states must be consistent with its transitions: if $\nabla(p, 1) = 1$ and $p \xrightarrow{\lceil_1} q$ then $\nabla(q, 1) = 2$ and $\nabla(q, i) = \nabla(p, i)$ for all $i \neq 1$.

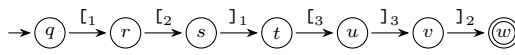
Language: $\{[1]_1 [1] [1]_1, [1]_1 [1]_1 [1] [1]_1,]_1 [1] [1]_1,]_1 [1] [1]_1\}$

Finite automaton:  no certificate

Regular expression: $e = ([1]_1 +]_1 [1]) [1] ([1]_1 + \varepsilon) \quad \nabla(e, 1) = 1 \quad \nabla^{\min}(e, 1) = -1$

(a) Unbalanced.

Language: $\{[1] [2]_1 [3]_3]_2\}$

Finite automaton: 

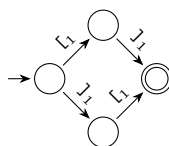
$\nabla(q, 1) = 0 \quad \nabla(r, 1) = 1 \quad \nabla(s, 1) = 1 \quad \nabla(t, 1) = 0$
 $\nabla(q, 2) = 0 \quad \nabla(r, 2) = 0 \quad \nabla(s, 2) = 1 \quad \nabla(t, 2) = 1 \quad \dots$
 $\nabla(q, 3) = 0 \quad \nabla(r, 3) = 0 \quad \nabla(s, 3) = 0 \quad \nabla(t, 3) = 0$

Regular expression: $e = [1] [2]_1 [3]_3]_2$

$\nabla(e, 1) = \nabla(e, 2) = \nabla(e, 3) = 0$
 $\nabla^{\min}(e, 1) = \nabla^{\min}(e, 2) = \nabla^{\min}(e, 3) = 0$

(b) Balanced.

Language: $\{[1]_1,]_1 [1]_1\}$

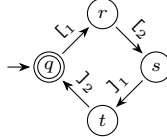
Finite automaton:  no certificate

Regular expression: $e = [1]_1 +]_1 [1]_1$

$\nabla(e, 1) = 0$
 $\nabla^{\min}(e, 1) = -1$

(c) Unbalanced.

Language: $\{[1] [2]_1]_2, [1] [2]_1]_2 [1] [2]_1]_2, \dots\}$

Finite automaton: 

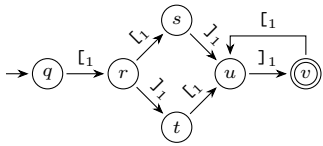
$\nabla(q, 1) = \nabla(t, 1) = 0$
 $\nabla(r, 1) = \nabla(s, 1) = 1$
 $\nabla(q, 2) = \nabla(r, 2) = 0$
 $\nabla(s, 2) = \nabla(t, 2) = 1$

Regular expression: $e = ([1] [2]_1]_2)^*$

$\nabla(e, 1) = \nabla(e, 2) = 0$
 $\nabla^{\min}(e, 1) = \nabla^{\min}(e, 2) = 0$

(d) Balanced.

Language: $\{[1] [1]_1]_1, [1]_1 [1]_1]_1, [1] [1]_1]_1 [1]_1, [1]_1 [1]_1]_1 [1]_1, \dots\}$

Finite automaton: 

$\nabla(q, 1) = \nabla(t, 1) = \nabla(v, 1) = 0$
 $\nabla(r, 1) = \nabla(u, 1) = 1$
 $\nabla(s, 1) = 2$

Regular expression: $e = [1] ([1]_1 +]_1 [1]_1) ([1]_1 [1]_1)^*]_1 \quad \nabla(e, 1) = 0 \quad \nabla^{\min}(e, 1) = 0$

(e) Balanced.

Figure 3.2: Examples of (un)balancedness.

Formally, we prove [Theorem 3.2](#).

Remark. The conditions above technically do not need to hold for unreachable or dead states, i.e., states unreachable from the initial state or states from which no final state can be reached. Since any such states can be ignored for the purpose of i -balances, we assume in the following that finite automata do not contain unreachable or dead states — unless an automaton’s language is empty, in which case we assume that it has exactly one state and no transitions.

Theorem 3.2. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton. Then M is balanced iff there exists a certificate $\nabla : Q \times \mathbb{N}^+ \rightarrow \mathbb{Z}$ such that, for all i :*

- $\nabla(q, i) \geq 0$ for all $q \in Q$;
- $\nabla(q, i) = 0$ for $q = q_0$ and for all $q \in F$; and
- if $(p, \sigma, q) \in \delta$ then $\nabla(q, i) = \nabla(p, i) + \nabla(\sigma, i)$, where $\nabla(\sigma, i) = 1$ if $\sigma = \llbracket_i$, $\nabla(\sigma, i) = -1$ if $\sigma = \rrbracket_i$ and $\nabla(\sigma, i) = 0$ otherwise.

Proof sketch. We prove the two directions separately. Given a certificate for M , we construct an equivalent regular expression and then use [Theorem 3.7](#) to show that the regular expression is balanced. Then, since they represent the same language, so is M . In the other direction, we construct a certificate for M and show that the construction can only fail if M is unbalanced. The full proof can be found in [Appendix A](#). \square

Example 3.3. [Figure 3.2](#) shows examples of balanced and unbalanced finite automata. The regular expressions in the figure can be ignored for now; we discuss them shortly.

For the automata in [Figures 3.2b](#), [3.2d](#) and [3.2e](#), a certificate exists, so they are balanced. In contrast, for the automata in [Figures 3.2a](#) and [3.2c](#), a certificate does not exist because it is impossible to assign a non-negative number to the bottom state of the diamond.

Regular expressions

We can apply the same notion of i -balance to regular expressions: to every i -balanced regular expression e , we assign a value $\nabla(e, i)$, which we show to correspond to the number n of unmatched opening i -parentheses in every word in its language. In general, for an arbitrary regular expression (e.g., $\varepsilon + \llbracket_i$

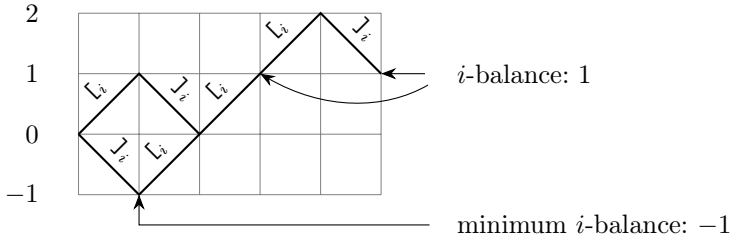


Figure 3.3: The i -balance and minimum i -balance of $([i]_i +]_i [i]_i [i]_i ([i]_i + \varepsilon))$. The i -balance points at the final height of the four words in the expression's language, which is the same in every case. The minimum i -balance points at the lowest height encountered by any word at any point.

and $[i^*$), n does not necessarily exist. In such cases we leave $\nabla(e, i)$ undefined. Shortly, we show that $\nabla(e, i)$ is defined for all e for which n exists.

Besides keeping track of the number of unmatched opening i -parentheses, additionally, we need to differentiate between, for example, $[i]_i$ and $]_i [i$. They have the same i -balance but the former is balanced while the latter is not. To do this we assign a second value which we call the *minimum i -balance*, denoted $\nabla^{\min}(e, i)$, which we show to correspond to the smallest i -balance among every prefix of every word in its language. Both are illustrated with a simple example in Figure 3.3. An expression is balanced if its i -balance and minimum i -balance equal 0 for every i . Note that, since ε is a prefix of every word and its i -balance is 0, the minimum i -balance can never be greater than 0.

Example 3.4. Figure 3.2 shows examples of balanced and unbalanced regular expressions, including their balances and minimum balances.

Formally, we define partial functions $\nabla, \nabla^{\min} : \mathbb{E} \times \mathbb{N}^+ \rightarrow \mathbb{Z}$ as in Figure 3.4. We show in Lemma 3.5 that ∇ and ∇^{\min} have the intended properties we described, and in Lemma 3.6 that they are defined when they should be.

As stated before, these functions are partial. In particular, $\nabla(e_1 + e_2, i)$ is defined only if $\nabla(e_1, i) = \nabla(e_2, i)$, while $\nabla(e^*, i)$ is defined only if $\nabla(e, i) = 0$. The definition of $\nabla^{\min}(e_1 \cdot e_2, i)$ relies on ∇ and may thus be undefined as well. The empty language \emptyset is a special case: we choose to leave $\nabla(\emptyset, i)$ and $\nabla^{\min}(\emptyset, i)$ undefined. Using standard algebraic rules, we can rewrite any regular expression representing a non-empty language into an equivalent expression

$$\begin{aligned}
\nabla(\llbracket_j, i) &= \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} & \nabla(\varepsilon, i) &= 0 \\
\nabla(\lceil_j, i) &= \begin{cases} -1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} & \nabla(e_1 \cdot e_2, i) &= \nabla(e_1, i) + \nabla(e_2, i) \\
& & \nabla(e_1 + e_2, i) &= \nabla(e_1, i) \quad \text{if } \nabla(e_1, i) = \nabla(e_2, i) \\
& & \nabla(e^*, i) &= 0 \quad \text{if } \nabla(e, i) = 0
\end{aligned}$$

(a) *i*-balance.

$$\begin{aligned}
\nabla^{\min}(\llbracket_j, i) &= 0 \\
\nabla^{\min}(\lceil_j, i) &= \begin{cases} -1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \\
\nabla^{\min}(\varepsilon, i) &= 0 \\
\nabla^{\min}(e_1 \cdot e_2, i) &= \min(\nabla^{\min}(e_1, i), \nabla(e_1, i) + \nabla^{\min}(e_2, i)) \\
\nabla^{\min}(e_1 + e_2, i) &= \min(\nabla^{\min}(e_1, i), \nabla^{\min}(e_2, i)) \\
\nabla^{\min}(e^*, i) &= \nabla^{\min}(e, i)
\end{aligned}$$

(b) Minimum *i*-balance.

Figure 3.4: The *i*-balance and minimum *i*-balance of regular expressions.

that does not contain \emptyset . To avoid overcomplicating our definitions and proofs, we assume for any regular expression e that e does not contain \emptyset , unless $e = \emptyset$.

Lemma 3.5. *Let $e \in \mathbb{E}$. If $\nabla(e, i)$ and $\nabla^{\min}(e, i)$ are defined, then:*

- (i) $|w|_{\llbracket_i} - |w|_{\lceil_i} = \nabla(e, i)$ for every $w \in L(e)$;
- (ii) $|v|_{\llbracket_i} - |v|_{\lceil_i} \geq \nabla^{\min}(e, i)$ for every prefix v of every $w \in L(e)$; and
- (iii) $|v|_{\llbracket_i} - |v|_{\lceil_i} = \nabla^{\min}(e, i)$ for some prefix v of some $w \in L(e)$.

Lemma 3.6. *Let $e \in \mathbb{E}$. If $|v|_{\llbracket_i} - |v|_{\lceil_i} = |w|_{\llbracket_i} - |w|_{\lceil_i}$ for every $v, w \in L(e)$ and $L(e) \neq \emptyset$, then $\nabla(e, i)$ and $\nabla^{\min}(e, i)$ are defined.*

The proofs are straightforward by structural induction on e . Applying [Lemmas 3.5](#) and [3.6](#) gives us the characterisation in [Theorem 3.7](#).

$$\begin{aligned}
 e &::= \emptyset \mid \varepsilon \mid [\cdot]_1 \mid [\cdot]_2 \mid \dots \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid e^* \\
 &\quad \mid \sqcup_{\theta}^1(e_1) \mid \sqcup_{\theta}^2(e_1, e_2) \mid \dots \\
 \theta &::= \emptyset \mid \varepsilon \mid 1 \mid 2 \mid \dots \mid \theta_1 + \theta_2 \mid \theta_1 \cdot \theta_2 \mid \theta^*
 \end{aligned}$$

Figure 3.5: A grammar \mathbb{E}^{\sqcup} for expressing balanced regular languages.

Theorem 3.7. *Let $e \in \mathbb{E}$. e is balanced iff $e = \emptyset$ or $\nabla(e, i) = \nabla^{\min}(e, i) = 0$, for every i .*

3.3 Balanced-by-construction regular languages

The main contribution of this section is a grammar of balanced-by-construction expressions, \mathbb{E}^{\sqcup} in Figure 3.5. We show that it can express all and only all balanced regular languages. It uses regular expressions as a basis and differs in two ways:

- Parentheses can syntactically occur only in ordered pairs instead of separately, so the atoms are all balanced.
- We add a family of operators $\sqcup_{\theta}^n(e_1, \dots, e_n)$, called *shuffle on trajectories*, in order to interleave words of subexpressions.

The shuffle on trajectories operator is a powerful variation of the traditional shuffle operator, which adds a control trajectory (or a set thereof) to restrict the permitted orders of interleaving. This allows for fine-grained control over orderings when shuffling words or languages. The binary operator was defined — and its properties thoroughly studied — by Mateescu et al. [MRS98]; we use the multiary variant, which was introduced slightly later [MSY00].

When defined on words, the shuffle on trajectories takes n words and a *trajectory*, which is a word over the alphabet $\{1, \dots, n\}$. This trajectory specifies the exact order of interleaving of the shuffled words: in Figure 3.6 the trajectory 1221112112 specifies that the result should first take a symbol from the first word, then from the second, then again from the second and so on.

Formally, let $w_1, \dots, w_n \in \Sigma^*$ and let $t \in \{1, \dots, n\}^*$. Then:

$$\sqcup_t^n(w_1, \dots, w_n) = \begin{cases} \sigma \sqcup_{t'}^n(w_1, \dots, w'_i, \dots, w_n) & \text{if } t = it' \wedge w_i = \sigma w'_i, \\ \varepsilon & \text{if } t = w_1 = \dots = w_n = \varepsilon. \end{cases}$$

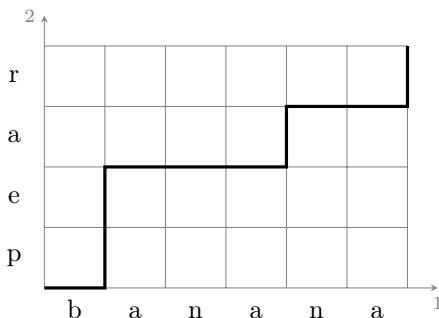


Figure 3.6: The shuffle of ‘banana’ and ‘pear’ over a trajectory 1221112112: ‘bpeanaanar’.

We note that $\sqcup_t^n(w_1, \dots, w_n)$ is only defined if the number of occurrences of i in t precisely matches the length of w_i , i.e., if $|t|_i = |w_i|$, for every i . If $|t|_i = |w_i|$, we say that t *fits* w_i .

Example 3.8.

- $\sqcup_{121332}([1]_1, [2]_2, [3]_3) = [1][2]_1[3]_3]_2$ since 121332 fits every word.
- $\sqcup_{121}^2([1]_1, [2]_2)$ is undefined since 121 does not fit $[2]_2$.

The shuffle on trajectories operator naturally generalises to languages and expressions: the shuffle of a number of languages on a set (i.e., language) of trajectories is defined as the set of all valid shuffles of words in the languages — valid meaning that the trajectory fits all the words, and thus that the result is defined. The language of a shuffle of expressions is the shuffle of the corresponding languages. Formally:

$$\begin{aligned} \sqcup_T^n(L_1, \dots, L_n) &= \{\sqcup_t^n(w_1, \dots, w_n) \mid t \in T, w_1 \in L_1, \dots, w_n \in L_n\}. \\ L(\sqcup_{L(\theta)}^n(e_1, \dots, e_n)) &= \sqcup_{L(\theta)}^n(L(e_1), \dots, L(e_n)). \end{aligned}$$

Example 3.9.

- $\sqcup_{12+21}^2([1 + [2]_2],]_1) \equiv [1]_1 +]_1[1$. Note that $[2]_2$ in the first operand is never used since no trajectory fits it.

- $\sqcup_{12+22}^2(\llbracket_1, \rrbracket_1) \equiv \llbracket_1 \rrbracket_1$. Note that the trajectory 22 is never used since it does not fit a word in either operand.
- $\sqcup_{(12)^*}^2((\llbracket_1 \rrbracket_1)^*, (\llbracket_2 \rrbracket_2)^*) \equiv (\llbracket_1 \llbracket_2 \rrbracket_1 \rrbracket_2)^*$.
- $\sqcup_{12+11}^2(\llbracket_1, \varepsilon) \equiv \emptyset$ since neither trajectory fits words in both operands simultaneously.
- $\sqcup_{(12)^*}^2(\llbracket_1 \rrbracket_1, \llbracket_2 \rrbracket_2 \rrbracket_2)^* \equiv \emptyset$ since no trajectory fits words in both operands simultaneously (due to a parity issue).

We say that a set of trajectories T fits L_i if every $t \in T$ fits some $w_i \in L_i$ and that θ fits e_i if $L(\theta)$ fits $L(e_i)$. We note that the grammar \mathbb{E}^\sqcup allows any regular set of trajectories.

As the operator's arity is clear from its operands, we will omit it from now on.

In the remainder of this section, we show that the grammar \mathbb{E}^\sqcup can express only (soundness) and all (completeness) balanced regular languages.

Soundness

Showing that every expression in \mathbb{E}^\sqcup represents a balanced regular language is straightforward. The base cases all comply and both balanced and regular languages are closed under choice, concatenation and finite repetition. The shuffle on trajectories operator yields an interleaving of its operands: a simple inductive proof will show closure of balanced languages under the operation. Mateescu et al. show that regular languages are closed under binary shuffle on regular trajectory languages by constructing an equivalent finite automaton [MRS98]; their construction can be generalised in a straightforward way to fit the multiary operator, which shows:

Theorem 3.10 (Soundness).

$$\{L(e) \mid e \in \mathbb{E}^\sqcup\} \subseteq \{L \mid L \text{ is a balanced and regular language}\}.$$

Completeness

To show that \mathbb{E}^\sqcup can express any balanced regular language, we take an arbitrary balanced regular expression e and show that there exists an equivalent expression in \mathbb{E}^\sqcup . We cannot use a simple inductive proof, as not every subexpression of a balanced regular expression is necessarily balanced; \llbracket_i and \rrbracket_i are

$$\begin{array}{ll}
\langle \rangle_i^k = ([_i]_i)^k ([_i]_i)^* & \langle \varepsilon \rangle_i^k = (\langle \rangle_i^k)^* \quad \langle \rangle_i^\omega = ([_i]_i)^\omega \\
\langle + \rangle_i^k = \langle \rangle_i^k [_i] & \langle - \rangle_i^k =]_i \langle \rangle_i^k \quad \langle \pm \rangle_i^k =]_i \langle \rangle_i^k [_i \quad \langle * \rangle_i^k = (\langle \pm \rangle_i^k)^* \quad \langle \pm \rangle_i^\omega = (]_i [_i)^\omega
\end{array}$$

Figure 3.7: Factors, with $i \in \mathbb{N}^+$ and $k \in \mathbb{N}$. The factors in the top row are balanced, those in the bottom row are not. We omit the superscript when it is not relevant. The ω -factors will be used in [Section 3.5](#).

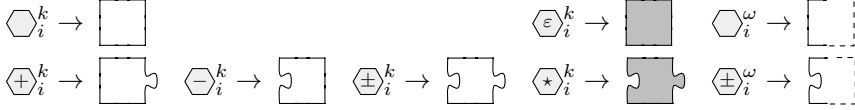


Figure 3.8: Factors as jigsaw pieces. A trailing opening parenthesis is visualised with a tab on the right side. A preceding closing parenthesis is visualised with a blank on the left side. Possible emptiness (in the case of repetition) is visualised by shading the piece. Infinite repetition is visualised by not finishing the piece on the right side.

the most obvious offenders. Instead, we show the more general result that, in fact, any regular expression whose every (minimum) i -balance is defined (but does not necessarily equal 0) can be written as the shuffle of a particular set of expressions, i.e., those in [Figure 3.7](#), which we call *factors*. Some of these factors are balanced and some are not; we show that the number of unbalanced factors can be limited by the expression's i -balance and minimum i -balance. In other words: if e is balanced then the resulting shuffle expression only uses balanced factors and is thus in \mathbb{E}^\sqcup .

To simplify matters, we consider only regular expressions that do not contain the operator '+'. After all, we can rewrite any regular expression into a disjunctive normal form $e_1 + \dots + e_n$ such that all e_i contain no '+'; since \mathbb{E}^\sqcup contains '+', as long as we can express any e_i , we can also express e . We thus do not have to consider '+' in our proofs. Additionally, we consider only regular expressions that do not contain the term ' \emptyset '. Again, we can rewrite any regular expression representing a nonempty language into an equivalent one that does not contain ' \emptyset ', without introducing any '+'. Since \mathbb{E}^\sqcup contains ' \emptyset ', all expressions representing the empty language are trivially covered and can thus be ignored.

For the remaining expressions we construct an equivalent expression $\sqcup_\theta(e_1, \dots, e_n)$, in such a way that the e_i are among the factors in [Figure 3.7a](#) and that the

$$\begin{array}{lll}
 (\langle \oplus_i^k, \ominus_i^\ell \rangle \rightarrow \langle \ominus_i^{k+\ell+1} & (\langle \ominus_i^k, \oplus_i^\ell \rangle \rightarrow \langle \oplus_i^{k+\ell} \\
 (\langle \oplus_i^k, \oplus_i^\ell \rangle \rightarrow \langle \oplus_i^{k+\ell+1} & (\langle \oplus_i^k, \ominus_i^\ell \rangle \rightarrow \langle \ominus_i^{k+\ell+1} \\
 (\langle \oplus_i^k, \star_i^\ell \rangle \rightarrow \langle \oplus_i^k & (\langle \star_i^k, \ominus_i^\ell \rangle \rightarrow \langle \ominus_i^\ell & (\langle \oplus_i^k, \oplus_i^\ell \rangle \rightarrow \langle \oplus_i^{k+\ell+1} \\
 (\langle \oplus_i^k, \star_i^\ell \rangle \rightarrow \langle \oplus_i^k & (\langle \star_i^\ell, \oplus_i^k \rangle \rightarrow \langle \oplus_i^k & (\langle \star_i^k, \star_i^\ell \rangle \rightarrow \langle \star_i^{\min(k,\ell)} \\
 (\langle \oplus_i^k, \oplus_i^\omega \rangle \rightarrow \langle \ominus_i^\omega & (\langle \oplus_i^k, \oplus_i^\omega \rangle \rightarrow \langle \oplus_i^\omega & (\langle \star_i^k, \oplus_i^\omega \rangle \rightarrow \langle \oplus_i^\omega
 \end{array}$$

Figure 3.9: Merging common pairs of factors, with $i \in \mathbb{N}^+$ and $k, \ell \in \mathbb{N}$.

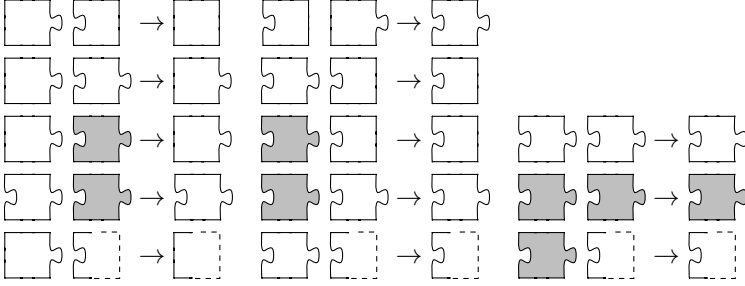




Figure 3.10: Merging pairs of factors as jigsaw pieces.

number of unbalanced factors depends on the original expression's (minimum) i -balances in a way that minimises it. We will use jigsaw pieces as a visual representation of the factors, where trailing opening parentheses and preceding closing parentheses are visualised with tabs and blanks, to make the construction and its proof more tangible. These are pictured in Figure 3.8.

Specifically, the jigsaw pieces help to understand a crucial step in our construction: we show that, under certain conditions, we can *merge* two operands of a shuffle, i.e., $\sqcup_T(L_1, \dots, L_{n-1}, L_n) = \sqcup_{T'}(L_1, \dots, L_{n-1}, L_n)$ for some T' . The formal details can be found in Lemma 3.11, where we define T' using morphisms, similarly to previous work [MRS98; Dom04]. In particular, we later use this lemma to merge the pairs of factors in Figure 3.9. In terms of jigsaw pieces, this merging operation may be thought of as fitting two pieces together; the aforementioned pairs of factors are shown using jigsaw pieces in Figure 3.10. In this visualisation, two factors can be merged if their pieces fit nicely together; typically, this consists of fitting a tab with a blank, which corresponds to matching a trailing opening parenthesis with a preceding closing one. The resulting pieces have the expected shape and are shaded only if both

original pieces were. The second pair of pieces in the top row of Figure 3.10 is a special case: these two are put back-to-back. There are more pieces that can be nicely put back-to-back: for example,  \rightarrow . However, since these other pairs are unnecessary for our later proof — unlike the ones in Figure 3.10 — they are omitted.




Lemma 3.11 (Merge). *Let $L = \sqcup_T(L_1, \dots, L_n)$. If*

- (a) *T fits every L_i ,*
- (b) *for every $t \in T$, if $t(i) = n - 1$ and $t(j) = n$ then $i < j$, and*
- (c) *for all $v, w \in L_{n-1}L_n$, if $|v| = |w|$ then $v = w$,*

then $L = \sqcup_{T'}(L_1, \dots, L_{n-1}L_n)$ for some T' such that T' fits $L_1, \dots, L_{n-1}L_n$.

Proof. Let ψ be a homomorphism such that $\psi(n) = n - 1$ and $\psi(i) = i$ for all other i . We proceed to show that $L = \sqcup_{\psi(T)}(L_1, \dots, L_{n-1}L_n)$. Since T fits every L_i , $\psi(T)$ also fits $L_1, \dots, L_{n-1}L_n$. \square

Note that we can reorder the operands of a shuffle as long as we also update the trajectories accordingly. Consequently, Lemma 3.11 essentially allows us to merge any two operands as long as the (transformed) conditions apply. We will generally use it as such.

Equipped with this merging lemma, we proceed with the construction. Recall that the result should be a single shuffle of factors, i.e., $\sqcup_\theta(e_1, \dots, e_n)$ for some factors e_1, \dots, e_n , with a minimal number of unbalanced factors. The construction is by structural induction. The base cases are straightforward: for ε , $[_i$ and $]_i$ we can use respectively $\sqcup_\varepsilon(\langle \diamond_i^0 \rangle)$, $\sqcup_1(\langle \oplus_i^0 \rangle)$ and $\sqcup_1(\langle \ominus_i^0 \rangle)$. In jigsaw pieces these are ,  and . Since we can ignore both ‘ \emptyset ’ and ‘+’, we are left with just two inductive cases: ‘.’ (concatenation) and ‘*’ (finite repetition).

- For concatenation, we use that $\sqcup_{T_1}(L_1, \dots, L_n) \cdot \sqcup_{T_2}(L_{n+1}, \dots, L_{n+m}) = \sqcup_{T_3}(L_1, \dots, L_{n+m})$ for some T_3 . Having combined the two into a single shuffle, we then apply Lemma 3.11 as needed to merge suitable pairs of factors to minimise the number of unbalanced factors. This is illustrated using jigsaw pieces in Figure 3.11. To understand the figure, suppose that we start with two expressions: the first with balance 1 and minimum balance -2 — for the sake of simplicity, there is only one type of parentheses — and the second with balance 0 and minimum balance -2 . In the figure, these two expressions are represented on the top row as

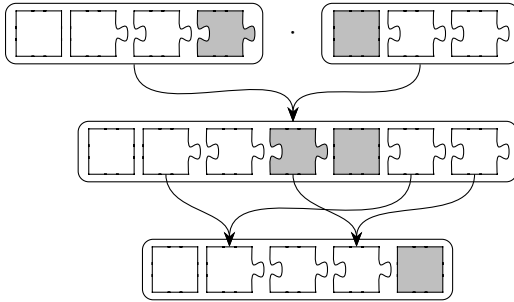


Figure 3.11: The concatenation (and merging) of two shuffles of factors as groups of jigsaw pieces.

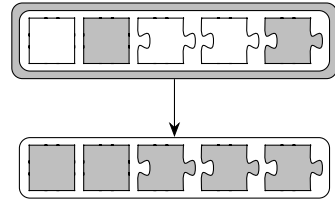


Figure 3.12: The finite repetition of a shuffle of factors as a group of jigsaw pieces.

two groups of jigsaw pieces; the number of blanks equals the minimum balance of the corresponding expression, while the number of tabs minus the number of blanks equals its balance. The first step is to combine the two groups into one. The other step is then to fit pairs of pieces from the first and the second group. Recall from Figure 3.4 that the balance of the resulting expression is 1 and its minimum balance is -2 ; indeed, the resulting group of pieces contains three tabs and two blanks. In other words, the resulting group of factors is minimal.

- For finite repetition, we show that, under our assumed conditions, $(\sqcup_T(L_1, \dots, L_n))^* = \sqcup_{T^*}(L_1^*, \dots, L_n^*)$. In jigsaw pieces, this simply means shading every non-shaded piece, as illustrated in Figure 3.12 — note that $(e^*)^* \equiv e^*$. The ‘*’ operator, visualised as an extra shaded container, does not change an expression’s balance or minimum balance, and since the number of tabs and blanks does not change either, the resulting group of factors remains minimal.

This construction and its proof are formalised in Lemma 3.12.

Lemma 3.12 (Rewrite). *Let $\text{pos}_i(e_1, \dots, e_n)$, $\text{neg}_i(e_1, \dots, e_n)$, $\text{neut}_i(e_1, \dots, e_n)$ be the number of \oplus_i , \ominus_i and $[\oplus_i \text{ or } \ominus_i]$ among e_1, \dots, e_n .*

Let $e \in \mathbb{E}$ such that e contains no ‘+’ and that its (minimum) i -balance is defined for every i . Then there exist θ and factors e_1, \dots, e_n such that $e \equiv \sqcup_\theta(e_1, \dots, e_n)$ and, additionally,

- (a) $\text{pos}_i(e_1, \dots, e_n) - \text{neg}_i(e_1, \dots, e_n) = \nabla(e, i)$ for every i ,
- (b) $-\text{neg}_i(e_1, \dots, e_n) - \text{neut}_i(e_1, \dots, e_n) = \nabla^{\min}(e, i)$ for every i ,
- (c) there are not both $\langle \oplus \rangle_i$ and $\langle \ominus \rangle_i$ among e_1, \dots, e_n for some i , and
- (d) θ fits every e_i .

Proof. This is a proof by induction on the structure of e .

The base cases ε , $[_i$ and $]_i$ are covered by $\sqcup_\varepsilon^1(\langle \ominus \rangle_i^0)$, $\sqcup_1^1(\langle \oplus \rangle_i^0)$ and $\sqcup_1^1(\langle \ominus \rangle_i^0)$. Since e contains no ‘+’, this leaves us with two inductive cases:

- Let $e = \hat{e}^*$. The induction hypothesis gives us $\hat{e}_1, \dots, \hat{e}_n$ and $\hat{\theta}$ satisfying all conditions for \hat{e} . It should be clear that $L((\sqcup_{\hat{\theta}}(\hat{e}_1, \dots, \hat{e}_n))^*) \subseteq L((\sqcup_{\hat{\theta}}(\hat{e}_1^*, \dots, \hat{e}_n^*))^*) \subseteq L(\sqcup_{\hat{\theta}^*}(\hat{e}_1^*, \dots, \hat{e}_n^*))$. Since $\nabla(e, i)$ is defined for all i , $\nabla(\hat{e}, i) = 0$ for all i . It then follows from (a) and (c) that $\hat{e}_1, \dots, \hat{e}_n$ contain no $\langle \oplus \rangle_i$ or $\langle \ominus \rangle_i$, so all \hat{e}_i^* are also factors.

To prove inclusion in the other direction, we show in two steps that $L(\sqcup_{\hat{\theta}^*}(\hat{e}_1^*, \dots, \hat{e}_n^*)) \subseteq L((\sqcup_{\hat{\theta}}(\hat{e}_1^*, \dots, \hat{e}_n^*))^*) \subseteq L((\sqcup_{\hat{\theta}}(\hat{e}_1, \dots, \hat{e}_n))^*)$.

The balances, minimum balances and factor counts are unchanged, so (a–c) are satisfied. Finally, since $\hat{\theta}$ fits every \hat{e}_i , $\hat{\theta}^*$ fits every \hat{e}_i^* , so (d) also holds.

- Let $e = \hat{e}_1 \cdot \hat{e}_2$. The induction hypothesis gives us some $e_{1,1}, \dots, e_{1,n_1}$ and θ_1 satisfying all conditions for \hat{e}_1 , and similarly for \hat{e}_2 . Let φ be a homomorphism such that $\varphi(i) = i + n_1$. Then $e' = \sqcup_{\theta_1 \varphi(\theta_2)}(e_{1,1}, \dots, e_{1,n_1}, e_{2,1}, \dots, e_{2,n_2}) \equiv e$ and e' satisfies (d), but not necessarily (a–c). We resolve the latter by merging operands $e_{1,j}, e_{2,k}$ where applicable with [Lemma 3.11](#).

We merge pairs of factors from [Figure 3.9](#), taking care to prioritise pairs containing both $\langle \oplus \rangle_i$ and $\langle \ominus \rangle_i$ over pairs containing only one of these, and pairs containing only one over pairs containing none. The first precondition of [Lemma 3.11](#) is assumed by induction; the second holds since we always take the first factor from e_1 and the second from e_2 ; the third holds for all pairs in [Figure 3.9](#). By [Lemma 3.11](#), the resulting expression is equivalent to e' and satisfies (d). It also satisfies (a–c). \square

Since a balanced regular expression has an i -balance and minimum i -balance of 0 for every i ([Theorem 3.7](#)), [Lemma 3.12](#) gives us [Theorem 3.13](#).

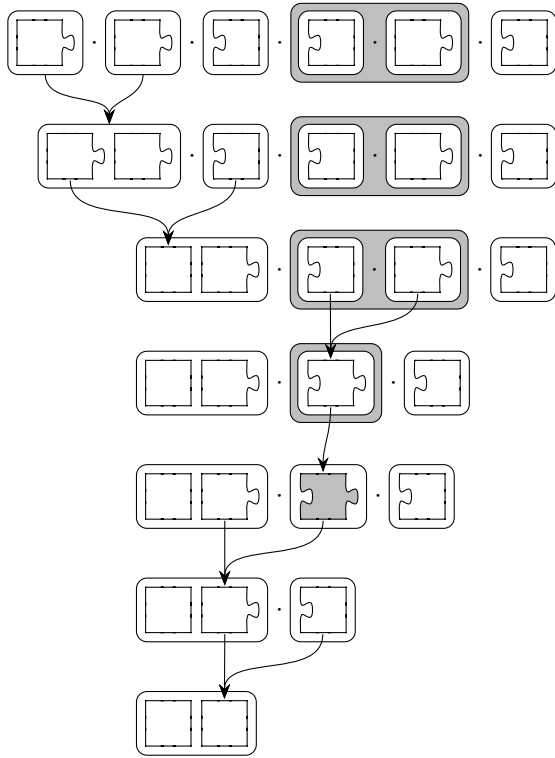


Figure 3.13: Construction of [Example 3.14](#) as groups of jigsaw pieces.

Theorem 3.13 (Completeness).

$$\{L(e) \mid e \in \mathbb{E}^{\sqcup}\} \supseteq \{L \mid L \text{ is a balanced and regular language}\}.$$

Example 3.14. Consider $e = [{}_1([{}_1]_1 +]_1[{}_1)(]_1[{}_1]^*)]_1$ ([Figure 3.2e](#)). We first rewrite e as $[{}_1[{}_1]_1(]_1[{}_1]^*)]_1 + [{}_1]_1[{}_1(]_1[{}_1]^*)]_1$. We proceed to show how to construct an expression in \mathbb{E}^{\sqcup} for the first part of the disjunction:

Boundedness

A (finite or infinite) word $w \in \Sigma^\infty$ is *i-bounded by* $n \in \mathbb{N}$ if $|v|_{\lceil_i} - |v|_{\rfloor_i} \leq n$ for all *finite* prefixes v of w . A language is *i-bounded by* n if all of its words are. A word or language is *i-bounded* if it is *i-bounded by* n for some n . A word or language is *bounded* if it is *i-bounded* for all i . The *minimal i-bound* of a word or language is the smallest n for which it is *i-bounded*. We extend these definitions to expressions and automata in the expected way: they are (*i*-)bounded if their corresponding languages are.

We note that the boundedness of words does not necessarily imply the boundedness of a language: all words in $[\ast_i(\lceil_i \rfloor_i)^\omega$ are bounded but the language itself is not.

Balancedness

A word $w \in \Sigma^\infty$ is *i-balanced* if $|w|_{\lceil_i} = |w|_{\rfloor_i}$, $|v|_{\lceil_i} \geq |v|_{\rfloor_i}$ for all (finite) prefixes v of w , and w is *i-bounded*. A language $L \subseteq \Sigma^\infty$ is *i-balanced* if all of its words are and if it is *i-bounded*. These are extended to balancedness and automata and expressions in the expected way: words and languages are balanced if they are *i-balanced* for all i , automata and expressions are (*i*-)balanced if their corresponding languages are. We note that all finite words are bounded by default, and that any regular language whose *i*-balance is defined is *i-bounded* as well. In other words: boundedness is only a restriction on infinite words and on languages containing infinite words.

Balanced ω -automata

Our characterisation of balanced ω -automata follows the same approach as for finite automata: we define a balance function and use this to specify balancedness conditions. We use two examples to illustrate the differences.

- Consider the automaton in [Figure 3.15](#). With acceptance condition $\{\{q_1, q_2\}, \{q_1, q_3\}, \{q_1, q_2, q_3\}\}$ it accepts the language $[\lceil_1(\lceil_1 \rfloor_1 + \lceil_2 \rfloor_2)^\omega$. The elements $\{q_1, q_2\}$ and $\{q_1, q_2, q_3\}$ both lead to the acceptance of balanced words, but $\{q_1, q_3\}$ yields unbalanced words: since q_2 is only visited finitely often, all resulting words have finite and unequal numbers of opening and closing 1-parentheses.

To classify this automaton as unbalanced, we thus require for every i and for every set of states in the acceptance condition either that there is some direct \lceil_i - or \rfloor_i -transition between two states in the set (violated in

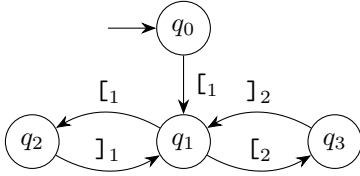


Figure 3.15: A Muller automaton with acceptance condition $\{\{q_1, q_2\}, \{q_1, q_3\}, \{q_1, q_2, q_3\}\}$, accepting the (unbalanced) language $[_1([_1]_1 + [_2]_2)^\omega$.

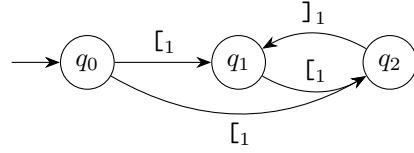


Figure 3.16: A Muller automaton with acceptance condition $\{\{q_1, q_2\}\}$, accepting the (balanced) language $(\varepsilon + [_1]([_1]_1)^\omega$.

Figure 3.15), in which case all corresponding words contain infinitely many i -parentheses of both kinds¹¹, or that the i -balance of all states in the set equals 0 (also violated in Figure 3.15), in which case all corresponding words contain finitely many i -parentheses and equally many opening and closing ones.

- Consider the automaton in Figure 3.16. Its language, $(\varepsilon + [_1]([_1]_1)^\omega$, is balanced, but no unique 1-balance can be assigned to states q_1 and q_2 : they can be either 1 and 2 if the first transition taken is that from q_0 to q_1 , or 0 and 1 if the first transition taken is that from q_0 to q_2 .

To remedy this, instead of assigning a single value as a state's i -balance, we now assign a range of values by giving an upper and lower bound on its i -balance. For the automaton in Figure 3.16, the lower and upper bound of the 1-balance of q_1 are respectively 0 and 1, and those of q_2 are respectively 1 and 2 — those of q_0 are both 0.

Formally, and combining these changes, we prove [Theorem 3.15](#).

Remark. Similar to finite automata, the conditions above do not need to hold for unreachable or dead states, where dead states in this case are states from which no state in one of the members of F can be reached, nor for members of F that specify an empty language. Since these can be ignored for the purpose of i -balance bounds, we assume in the following that Muller automata do not contain unreachable or dead states, and that every member of F specifies a

¹¹Since the acceptance condition of Muller automata defines cyclic subgraphs (for nonempty languages), the presence of one implies the presence of the other. If not, it would be impossible to define a unique balance for all states.

nonempty language — unless the automaton’s language is empty, in which case we assume that it has one state and no transitions, and that F is empty.

Theorem 3.15. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a Muller automaton. M is balanced iff there exist lower and upper bounds on the i -balance of every state in Q (respectively $\nabla^L(q, i)$ and $\nabla^U(q, i)$ for a state q) such that, for every i ,*

- (i) $\nabla^L(q_0, i) = \nabla^U(q_0, i) = 0$;
- (ii) $\nabla^L(q, i) \geq 0$ for every $q \in Q$;
- (iii) $\nabla^L(q, i) \leq \nabla^L(p, i) + \nabla(\sigma, i) \leq \nabla^U(p, i) + \nabla(\sigma, i) \leq \nabla^U(q, i)$ for every $(p, \sigma, q) \in \delta$; and
- (iv) for every $\{f_1, \dots, f_\ell\} \in F$ (representing a nonempty language), either there exist some j, k such that $(\nabla^L(f_j, i), \nabla^U(f_j, i)) \neq (\nabla^L(f_k, i), \nabla^U(f_k, i))$, or $\nabla^L(f_j, i) = \nabla^U(f_j, i) = 0$ for every j .

Proof sketch. We prove the two directions separately. Given bounds satisfying the conditions in the theorem, we construct an equivalent ω -regular expression and then use [Theorem 3.19](#) to show that it — and thus M — is balanced. In the other direction, we construct these bounds and then show that they satisfy the four conditions. The full proof can be found in [Appendix A](#). \square

Balanced ω -regular expressions

Our characterisation of balanced ω -regular expressions is a generalisation of that of balanced regular expressions, adapted to deal with the complications noted with ω -automata:

- We introduce two predicates for expressions: $\xi(e, i)$ and $\xi^\omega(e, i)$. Intuitively, $\xi(e, i)$ iff every word in $L(e)$ contains at least one i -parenthesis, while $\xi^\omega(e, i)$ iff every word in $L(e)$ contains infinitely many. The predicates are formally defined in [Figure 3.17](#), while the corresponding properties are shown in [Lemma 3.16](#).
- As with ω -automata, we swap single i -balances for pairs of lower and upper bounds $\nabla^L(e, i)$ and $\nabla^U(e, i)$. These bounds are formally defined in [Figure 3.18](#). The changes to minimum i -balance are as expected: we add $\nabla^{\min}(e^\omega, i) = \nabla^{\min}(e, i)$ and we redefine $\nabla^{\min}(e_1 \cdot e_2, i)$ to use $\nabla^L(e_1, i)$

$$\begin{array}{c}
\overline{\xi(\llbracket i, i \rrbracket)} \quad \overline{\xi(\llbracket i, i \rrbracket)} \quad \frac{\xi(e_j, i) \quad j \in \{1, 2\}}{\xi(e_1 \cdot e_2, i)} \quad \frac{\xi(e_1, i) \quad \xi(e_2, i)}{\xi(e_1 + e_2, i)} \quad \frac{\xi(e, i)}{\xi(e^\omega, i)} \\
\frac{\xi(e, i)}{\xi^\omega(e^\omega, i)} \quad \frac{\xi^\omega(e_2, i)}{\xi^\omega(e_1 \cdot e_2, i)} \quad \frac{\xi^\omega(e_1, i) \quad \xi^\omega(e_2, i)}{\xi^\omega(e_1 + e_2, i)}
\end{array}$$

Figure 3.17: The i -occurrence of regular and ω -regular expressions.

$$\begin{aligned}
\nabla^\dagger(\llbracket i, j \rrbracket) &= \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \\
\nabla^\dagger(\llbracket i, j \rrbracket) &= \begin{cases} -1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \\
\nabla^\dagger(\varepsilon, i) &= 0 \\
\nabla^\dagger(e_1 \cdot e_2, i) &= \begin{cases} \nabla^\dagger(e_2, i) & \text{if } \xi^\omega(e_2, i) \\ \nabla^\dagger(e_1, i) + \nabla^\dagger(e_2, i) & \text{otherwise} \end{cases} \\
\nabla^L(e_1 + e_2, i) &= \min(\nabla^L(e_1, i), \nabla^L(e_2, i)) \\
\nabla^U(e_1 + e_2, i) &= \max(\nabla^L(e_1, i), \nabla^L(e_2, i)) \\
\nabla^\dagger(e^*, i) &= 0 \quad \text{if } \nabla^\dagger(e, i) = 0 \\
\nabla^\dagger(e^\omega, i) &= 0 \quad \text{if } \nabla^\dagger(e, i) = 0
\end{aligned}$$

Figure 3.18: The i -balance lower and upper bounds of regular and ω -regular expressions, where $\dagger \in \{L, U\}$.

instead of $\nabla(e_1, i)$. The other equations remain as in [Figure 3.4](#). The corresponding properties are shown in [Lemmas 3.17](#) and [3.18](#). We note that, for any regular expression $e \in \mathbb{E}$, $\nabla^L(e, i) = \nabla^U(e, i) = \nabla(e, i)$.

As before, we assume without loss of generality that an ω -regular expression e does not contain the term ‘ \emptyset ’, unless $e = \emptyset$, to simplify our definitions and proofs.

Lemma 3.16. *Let $e \in \mathbb{E} \cup \Omega$. If $e \neq \emptyset$, then:*

- (i) $\xi(e, i)$ iff $|w|_{\lceil_i} + |w|_{\rfloor_i} > 0$ for every $w \in L(e)$;
- (ii) $\xi^\omega(e, i)$ iff $|w|_{\lceil_i} + |w|_{\rfloor_i} = \aleph_0$ for every $w \in L(e)$.

Lemma 3.17 (cf. Lemma 3.5). *Let $e \in \mathbb{E} \cup \Omega$. If $\nabla^L(e, i)$, $\nabla^U(e, i)$ and $\nabla^{\min}(e, i)$ are defined, then:*

- (i) For every $w \in L(e)$, $|w|_{\lceil_i}$ and $|w|_{\rfloor_i}$ are either both finite or both infinite;
- (ii) For every $w \in L(e)$, if $|w|_{\lceil_i}, |w|_{\rfloor_i}$ are finite, then $\nabla^L(e, i) \leq |w|_{\lceil_i} - |w|_{\rfloor_i} \leq \nabla^U(e, i)$;
- (iii) If $e \in \mathbb{E}$, then there exist $w_1, w_2 \in L(e)$ such that $|w_1|_{\lceil_i} - |w_1|_{\rfloor_i} = \nabla^L(e, i)$ and $|w_2|_{\lceil_i} - |w_2|_{\rfloor_i} = \nabla^U(e, i)$;
- (iv) If $\xi^\omega(e, i)$, then $\nabla^L(e, i) = \nabla^U(e, i) = 0$;
- (v) $|v|_{\lceil_i} - |v|_{\rfloor_i} \geq \nabla^{\min}(e, i)$ for every finite prefix v of every $w \in L(e)$;
- (vi) $|v|_{\lceil_i} - |v|_{\rfloor_i} = \nabla^{\min}(e, i)$ for some finite prefix v of some $w \in L(e)$;
- (vii) $L(e)$ is i -bounded.

Lemma 3.18 (cf. Lemma 3.6). *Let $e \in \mathbb{E} \cup \Omega$. If $e \neq \emptyset$, if e is i -bounded, and if there exists some n such that $|(v|_{\lceil_i} - |v|_{\rfloor_i}) - (w|_{\lceil_i} - |w|_{\rfloor_i})| \leq n$ for all $v, w \in L(e)$ with finite i -parenthesis counts, then $\nabla^L(e, i)$, $\nabla^U(e, i)$ and $\nabla^{\min}(e, i)$ are defined.*

The proofs of these lemmas are straightforward by structural induction on e . Applying these lemmas gives us the following characterisation:

Theorem 3.19. *Let $e \in \mathbb{E} \cup \Omega$. Then e is balanced iff $\nabla^L(e, i) = \nabla^U(e, i) = \nabla^{\min}(e, i) = 0$ for every i or if $e = \emptyset$.*

Example 3.20. In this example, we further illustrate the role of lower and upper bounds on i -balances in the presence of ω .

- Let $e_1 = \lceil_1 + \lceil_1 \lceil_1 \lceil_1$. We have:

$$\begin{aligned}\nabla^L(\lceil_1 + \lceil_1 \lceil_1 \lceil_1, 1) &= \min(\nabla^L(\lceil_1, 1), \nabla^L(\lceil_1 \lceil_1 \lceil_1, 1)) = \min(1, 3) = 1 \\ \nabla^U(\lceil_1 + \lceil_1 \lceil_1 \lceil_1, 1) &= \max(\nabla^U(\lceil_1, 1), \nabla^U(\lceil_1 \lceil_1 \lceil_1, 1)) = \max(1, 3) = 3\end{aligned}$$

Thus, e_1 is unbalanced. Intuitively, the problem is that there are unmatched opening 1-parentheses.

- Let $e_2 = e_1 \cdot (\lceil_1 \lceil_1)^*$. Since $\neg\xi^\omega((\lceil_1 \lceil_1)^*, 1)$, we have:

$$\begin{aligned}\nabla^L(e_1 \cdot (\lceil_1 \lceil_1)^*, 1) &= \nabla^L(e_1, 1) + \nabla^L((\lceil_1 \lceil_1)^*, 1) = 1 + 0 = 1 \\ \nabla^U(e_1 \cdot (\lceil_1 \lceil_1)^*, 1) &= \nabla^U(e_1, 1) + \nabla^U((\lceil_1 \lceil_1)^*, 1) = 3 + 0 = 3\end{aligned}$$

Thus, e_2 is unbalanced. Intuitively, the problem remains that there are unmatched opening 1-parentheses.

- Let $e_3 = e_1 \cdot (\lceil_1 \lceil_1)^\omega$. Since $\xi^\omega((\lceil_1 \lceil_1)^\omega, 1)$, we have:

$$\begin{aligned}\nabla^L(e_1 \cdot (\lceil_1 \lceil_1)^\omega, 1) &= \nabla^L((\lceil_1 \lceil_1)^\omega, 1) = 0 \\ \nabla^U(e_1 \cdot (\lceil_1 \lceil_1)^\omega, 1) &= \nabla^U((\lceil_1 \lceil_1)^\omega, 1) = 0\end{aligned}$$

Thus, e_3 is balanced, even though its prefix e_1 is unbalanced. Intuitively, the solution is that even though, infinitely often, there are more opening parentheses than closing parentheses, every opening parenthesis is eventually matched and the number of unmatched opening parentheses never exceeds three. This precisely coincides with the notion of balancedness for infinite words that we adopted, as motivated at the beginning of this section.

3.5 Balanced-by-construction ω -regular languages

To construct balanced ω -regular expressions, we extend the grammar in [Figure 3.5](#) with ω as in [Definition 2.3](#) to obtain the expression grammar Ω^\sqcup in [Figure 3.19](#).

Since the inductive definition of the shuffle on trajectories operator does not support words of infinite length, we formally redefine it as follows. The definition is adapted from the definition given by Mateescu et al. for the binary case [[MRS98](#)]. Let $w_1, \dots, w_n \in \Sigma^\infty$ and let $t \in \{1, \dots, n\}^\infty$. If t fits w_1, \dots, w_n , i.e., if $|t|_i = |w_i|$ for every i , then $\sqcup_t(w_1, \dots, w_n) = w(1)w(2) \dots w(|t|)$ if t has

$$\begin{aligned}
e &::= \emptyset \mid e + e \mid E \cdot e \mid E_+^\omega \mid \sqcup_{T_\omega} (C, \dots, C) && (\omega\text{-regular}) \\
E &::= \emptyset \mid \varepsilon \mid P \mid E + E \mid E \cdot E \mid E^* \mid \sqcup_T (E, \dots, E) && (\text{regular}) \\
E_+ &::= \emptyset \mid P \mid E_+ + E_+ \mid E \cdot E_+ \cdot E \mid \sqcup_{T_+} (E, \dots, E) && (\text{non-empty}) \\
P &::= [_1 \cdot]_1 \mid [_2 \cdot]_2 \mid \dots && (\text{parentheses}) \\
C &::= e \mid E && (\omega\text{-shuffle operand}) \\
T &::= \emptyset \mid \varepsilon \mid 1 \mid 2 \mid \dots \mid T + T \mid T \cdot T \mid T^* && (\text{trajectory}) \\
T_+ &::= \emptyset \mid 1 \mid 2 \mid \dots \mid T_+ + T_+ \mid T \cdot T_+ \cdot T && (\text{non-empty}) \\
T_\omega &::= \emptyset \mid T_\omega + T_\omega \mid T \cdot T_\omega \mid T_+^\omega && (\omega\text{-trajectory})
\end{aligned}$$

Figure 3.19: A grammar Ω^ω for expressing balanced ω -regular languages.

finite length and $w(1)w(2)\dots$ if t has infinite length, where $w(i) = w_j(k)$ for $j = t(i)$ and $k = |t(1, i)|_j$. The result is as expected. As before, this naturally extends to languages and expressions.

Soundness

Balanced languages are closed under shuffle on trajectories; this follows immediately from the operator's definition. Mateescu et al. show that ω -regular languages are closed under binary shuffle on ω -regular trajectory languages [MRS98]. We extend their result to multiary shuffles by constructing a Muller automaton M for $\sqcup_T(L_1, \dots, L_n)$ out of a Muller automaton M_T for T and finite/Muller automata M_1, \dots, M_n for L_1, \dots, L_n , where L_1, \dots, L_n are all either regular or ω -regular.

The construction of M is analogous to the construction of a finite automaton for a shuffle of regular languages and differs only in the construction of the acceptance criterion F . To define it, let F_i be the acceptance criterion of M_i : if M_i is a finite automaton, then F_i is a set of states, and we may assume without loss of generality that no state in F_i has any outgoing transition (otherwise, we can construct an equivalent automaton which does have this property); if M_i is a Muller automaton, then F_i is a set of sets of states. We also assume, without loss of generality, that every i such that L_i is ω -regular occurs infinitely often in every trajectory in T .

We now define F as a combination of all the F_i : F is the set of sets of states such that, if L_i is ω -regular, then the projection of these states on i is an element

of F_i , and if L_i is regular, then the projection of these states on i is a single state in F_i . Formally, let $\varphi_i((q_t, q_1, \dots, q_n)) = q_i$ and $\varphi_i(S) = \{\varphi_i(q) \mid q \in S\}$ be the projection of a state or set of states on i . Then $F = \{S \mid S \subseteq Q \wedge \forall i \in [1, n] : (\varphi_i(S) \in F_i \vee (\varphi_i(S) \subseteq F_i \wedge |\varphi_i(S)| = 1))\}$, where Q is the set of states of M . The automaton for T forces that every Muller automaton for some L_i takes infinitely many steps. By our assumption that the final states of finite automata have no outgoing transitions, all finite automata only take a finite number of steps. It follows that our constructed Muller automaton accepts the language of $\sqcup_T(L_1, \dots, L_n)$, which then must be ω -regular. In other words:

Theorem 3.21 (Soundness).

$$\{L(e) \mid e \in \Omega^\sqcup\} \subseteq \{L \mid L \text{ is a balanced } \omega\text{-regular language}\}.$$

Completeness

Our approach to showing that every balanced ω -regular expression has an equivalent expression in Ω^\sqcup mirrors that of [Section 3.3](#): we first rewrite an expression into a disjunctive normal form and then recursively construct an expression in Ω^\sqcup for every term of the disjunction by merging pairs of factors.

Let $e \neq \emptyset$ be a balanced ω -regular expression. Without loss of generality, we may assume that $e = e_1 e_2^\omega + \dots + e_{2m-1} e_{2m}^\omega$, where every e_i is a regular expression containing no ‘+’ or ‘ \emptyset ’ [[McN66](#)]. Otherwise, we can rewrite it accordingly. We show how to construct an expression in Ω^\sqcup for $e_1 e_2^\omega$.

Since $\nabla^L(e, i) = \nabla^U(e, i) = \nabla^{\min}(e, i) = 0$ by [Theorem 3.19](#), it follows that $\nabla^{\min}(e_1, i) = \nabla^L(e_2, i) = \nabla^U(e_2, i) = 0$. Then, by [Lemma 3.12](#), we can write e_1 as a shuffle of factors $\langle \rangle_i, \langle \varepsilon \rangle_i, \langle \oplus \rangle_i$ and e_2 as a shuffle of factors $\langle \rangle_i, \langle \varepsilon \rangle_i, \langle \oplus \rangle_i, \langle \star \rangle_i$. The idea is then to: (a) rewrite e_2^ω in terms of factors $\langle \rangle_i, \langle \varepsilon \rangle_i, \langle \oplus \rangle_i^\omega, \langle \oplus \rangle_i^\omega$ and then; (b) merge every $\langle \oplus \rangle_i$ in e_1 with a $\langle \oplus \rangle_i^\omega$ in e_2^ω into $\langle \oplus \rangle_i^\omega$, using [Lemma 3.11](#). We run into two complications:

- In step (a), e_2^ω may not necessarily be expressible as a single shuffle of factors. Consider $e_2 = [1]_1([2]_2)^* \equiv \sqcup_{11(22)^*}(\langle \rangle_1, \langle \varepsilon \rangle_2)$: then e_2^ω contains both words with finite and infinite numbers of 2-parentheses. The latter requires a factor $\langle \rangle_2^\omega$, while the former requires its absence.

To remedy this, we write e_2^ω as a *disjunction* of shuffles of factors; one for

every combination of finite and infinite versions of $\langle \ominus \rangle_i, \langle \varepsilon \rangle_i$. In this case:

$$\begin{aligned} e_2^\omega &\equiv \sqcup_{(11(22)^*)^\omega} (\langle \ominus \rangle_1, \langle \varepsilon \rangle_2) \\ &\quad + \sqcup_{(11(22)^*)^\omega} (\langle \ominus \rangle_1^\omega, \langle \varepsilon \rangle_2) \\ &\quad + \sqcup_{(11(22)^*)^\omega} (\langle \ominus \rangle_1, \langle \ominus \rangle_2^\omega) \\ &\quad + \sqcup_{(11(22)^*)^\omega} (\langle \ominus \rangle_1^\omega, \langle \ominus \rangle_2^\omega) \end{aligned}$$

The second term of the disjunction contains all words with finitely many 2-parentheses and the fourth term contains all those with infinitely many. Note that the first and third terms yield empty languages, but this does not affect the resulting language. It is thus unnecessary to exclude these.

This is further detailed in [Lemma 3.22](#).

- In step (b), the number of $\langle \oplus \rangle_i^\omega$ in a term of e_2^ω may not necessarily match the number of $\langle \oplus \rangle_i$ in e_1 : if $e_1 = []_1$ and $e_2 = []_1$, then e_1 contains one $\langle \oplus \rangle_1$ and e_2 contains one factor $\langle \ominus \rangle_1$. To solve this, we use two observations:

- We can apply [Lemma 3.11](#) to split a $\langle \ominus \rangle_i$ into $\langle \oplus \rangle_i$ and $\langle \ominus \rangle_i$, inverting the direction in which we have used the lemma so far.
- Since $e_2^\omega \equiv (e_2 \cdot e_2)^\omega$, we can produce arbitrarily many copies of the factors in e_2 .

Combining the above, we can always split a $\langle \ominus \rangle_i$ into $\langle \oplus \rangle_i$ and $\langle \ominus \rangle_i$, then create copies of them and merge them back into one $\langle \ominus \rangle_i$ and one $\langle \oplus \rangle_i$. Since we can merge all other factors with their own copy, this effectively adds one $\langle \oplus \rangle_i$. Now that we have at least one, we can create more: we create a copy of every factor, then merge every factor with its own copy except for some number of $\langle \oplus \rangle_i$. This is further detailed in [Lemma 3.23](#).

Lemma 3.22. *Let $e = \sqcup_\theta(e_1, \dots, e_n) \in \mathbb{E}^\sqcup$ be a shuffle of factors $\langle \ominus \rangle_i, \langle \varepsilon \rangle_i, \langle \oplus \rangle_i$ such that θ fits every e_j and contains no $+$. Then $e^\omega \equiv \hat{e}_1 + \dots + \hat{e}_m$, where $\hat{e}_k = \sqcup_{\theta_k}(e_{k,1}, \dots, e_{k,n})$ is a shuffle of factors $\langle \ominus \rangle_i, \langle \varepsilon \rangle_i, \langle \ominus \rangle_i^\omega, \langle \oplus \rangle_i^\omega$ for every k such that the number of $\langle \oplus \rangle_i$ in e is the same as the number of $\langle \oplus \rangle_i^\omega$ in \hat{e}_k for every i , and θ_k fits every $e_{k,j}$.*

Proof. Let $\varphi : \mathbb{E} \rightarrow 2^{\mathbb{E} \cup \Omega}$ such that $\varphi(\langle \ominus \rangle_i^k) = \{\langle \ominus \rangle_i^k, \langle \ominus \rangle_i^\omega\}$, $\varphi(\langle \varepsilon \rangle_i^k) = \{\langle \varepsilon \rangle_i^k, \langle \ominus \rangle_i^\omega\}$ and $\varphi(\langle \oplus \rangle_i^k) = \{\langle \oplus \rangle_i^\omega\}$. We can then show that $e^\omega \equiv \hat{e}_1 + \dots + \hat{e}_m$, where $\{\hat{e}_1, \dots, \hat{e}_m\} = \{\sqcup_{\theta^\omega}(e'_1, \dots, e'_n) \mid e'_1 \in \varphi(e_1), \dots, e'_n \in \varphi(e_n)\}$.

Moreover, since φ maps $\langle \oplus \rangle_i$ to $\langle \oplus \rangle_i^\omega$, the number of factors $\langle \oplus \rangle_i^\omega$ in every \hat{e}_k matches the number of factors $\langle \oplus \rangle_i$ in e . However, if $\hat{e}_k = \sqcup_{\theta^\omega}(e'_1, \dots, e'_n)$, then θ^ω may not necessarily fit every e'_j : if e'_j is one of $\langle \ominus \rangle_i, \langle \varepsilon \rangle_i$, then there are $t \in L(\theta^\omega)$ with infinitely many j , while every word in $L(e'_j)$ is finite. Instead of θ^ω , we can use the trajectory $\theta^* \cdot \psi(\theta)^\omega$, where ψ is a homomorphism such that $\psi(j) = \varepsilon$ if e'_j is one of $\langle \ominus \rangle_i, \langle \varepsilon \rangle_i$ and $\psi(j) = j$ otherwise. This covers exactly the part of θ^ω that fits every e'_j . \square

Lemma 3.23. *Let $\sqcup_\theta(e_1, \dots, e_n) \equiv e \in \mathbb{E}$ be a shuffle of factors $\langle \ominus \rangle_i, \langle \varepsilon \rangle_i, \langle \oplus \rangle_i, \langle \star \rangle_i$ such that θ fits every e_j and contains no $+$, and $\xi(e, i)$. If there are ℓ factors $\langle \oplus \rangle_i, \langle \star \rangle_i$ among e_1, \dots, e_n , then for every $k \geq \ell$ (such that $k > 0$), there exists some shuffle of factors $\hat{e} = \sqcup_{\hat{\theta}}(\hat{e}_1, \dots, \hat{e}_m)$ such that $e^\omega \equiv \hat{e}^\omega$, \hat{e} contains k factors $\langle \oplus \rangle_i$ and no $\langle \star \rangle_i$ and $\hat{\theta}$ fits every \hat{e}_j .*

Proof. This proof consists of three steps. First, we need to make sure that we have at least one $\langle \oplus \rangle_i$. Second, we replace any remaining factors $\langle \star \rangle_i$ with $\langle \oplus \rangle_i$. Third, we create additional copies of $\langle \oplus \rangle_i$ as needed.

1. Suppose that there are no $\langle \oplus \rangle_i$ among e_1, \dots, e_n . Then our first step consists of creating one. Since $\xi(e, i)$ and θ contains no $+$, there exists some $e_j \in \{\langle \ominus \rangle_i, \langle \varepsilon \rangle_i, \langle \star \rangle_i\}$ such that $|t|_j > 0$ for every $t \in L(\theta)$. Without loss of generality, we may assume that $j = n$.

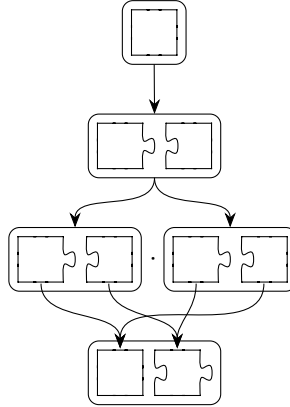
If $e_n = \langle \star \rangle_i^k$, since $|t|_n > 0$ for every t then $e \equiv \sqcup_\theta(e_1, \dots, \langle \oplus \rangle_i^k)$ and we can proceed with step 2. Otherwise, if $e_n = \langle \varepsilon \rangle_i^k$, then $e \equiv \sqcup_\theta(e_1, \dots, \langle \ominus \rangle_i^k)$ and if $e_n = \langle \ominus \rangle_i^k$, then $e \equiv \sqcup_\theta(e_1, \dots, \langle \oplus \rangle_i^1)$. Going forward, we may thus assume that $e_n = \langle \oplus \rangle_i^k$ with $k \geq 1$. Since $|t|_n > 0$ for every $t \in L(\theta)$ and θ contains no $+$, it follows that $\theta = \theta_1 \cdot \theta_2$ such that both θ_1 and θ_2 only contain trajectories with odd numbers of n . We can then apply the proof of [Lemma 3.11](#) to show that $e \equiv \sqcup_{\theta_3}(e_1, \dots, e_{n-1}, \langle \oplus \rangle_i^{k_1}, \langle \oplus \rangle_i^{k_2})$ for some θ_3, k_1, k_2 .

If e_1, \dots, e_{n-1} contain a $\langle \star \rangle_i$, then without loss of generality we may assume that $e_{n-1} = \langle \star \rangle_i^{k_3}$. We may assume that there exists some $t \in L(\theta)$ such that $|t|_{n-1} = 0$; otherwise we would have selected this factor as e_n earlier in this step and then proceeded with step 2. It follows that all trajectories in θ_1 and θ_2 , and therefore in θ_3 , contain even numbers of n . Then, in the same way that we split $\langle \oplus \rangle_i^k$ into $\langle \oplus \rangle_i^{k_1}$ and $\langle \oplus \rangle_i^{k_2}$

before, we can show that $e \equiv \sqcup_{\theta_4}(e_1, \dots, e_{n-2}, \langle \star \rangle_i^{k_4}, \langle \star \rangle_i^{k_5}, \langle \oplus \rangle_i^{k_1}, \langle \ominus \rangle_i^{k_2})$ for some θ_4, k_4, k_5 . As seen in [Figure 3.9](#), we can then merge $\langle \star \rangle_i^{k_4}$ with $\langle \ominus \rangle_i^{k_2}$ and $\langle \star \rangle_i^{k_5}$ with $\langle \oplus \rangle_i^{k_1}$ to obtain $e \equiv \sqcup_{\theta_5}(e_1, \dots, e_{n-2}, \langle \oplus \rangle_i^{k_1}, \langle \ominus \rangle_i^{k_2})$ for some θ_5 . This covers the case where $k = \ell > 0$ but there are no factors $\langle \oplus \rangle_i$. We may thus assume without loss of generality that $e \equiv \sqcup_{\theta_6}(e_1, \dots, \langle \oplus \rangle_i^{k_1}, \langle \ominus \rangle_i^{k_2})$ for some θ_6 .

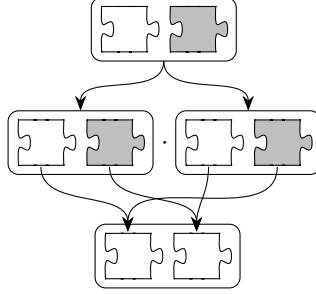
Since we still lack a $\langle \oplus \rangle_i$, we use that $e^\omega \equiv (e \cdot e)^\omega$ to construct $e' = \sqcup_{\theta_6}(e_1, \dots, \langle \oplus \rangle_i^{k_1}, \langle \ominus \rangle_i^{k_2}) \cdot \sqcup_{\theta_6}(e_1, \dots, \langle \oplus \rangle_i^{k_1}, \langle \ominus \rangle_i^{k_2}) \equiv \sqcup_{\theta_7}(e_1, \dots, \langle \oplus \rangle_i^{k_1}, \langle \ominus \rangle_i^{k_2}, e_1, \dots, \langle \oplus \rangle_i^{k_1}, \langle \ominus \rangle_i^{k_2})$ for some θ_7 . We can then merge the first $\langle \oplus \rangle_i^{k_1}$ with the second $\langle \ominus \rangle_i^{k_2}$ into $\langle \ominus \rangle_i^{k_1+k_2+1}$ and merge the second $\langle \oplus \rangle_i^{k_1}$ with the first $\langle \ominus \rangle_i^{k_2}$ into $\langle \oplus \rangle_i^{k_1+k_2}$. We can merge every other factor with its own copy, which gives us $e' \equiv \sqcup_{\theta_8}(e'_1, \dots, \langle \ominus \rangle_i^{k_1+k_2+1}, \langle \oplus \rangle_i^{k_1+k_2})$ and $e_1^\omega \equiv e^\omega$.

Not including corner cases, this step can be visualised as follows:



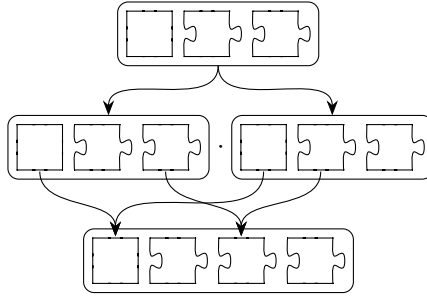
- Now that we have at least one $\langle \oplus \rangle_i$, we can reuse methods applied in the first step to replace any remaining $\langle \star \rangle_i$: create a copy of every factor using $e^\omega \equiv (e \cdot e)^\omega$, then merge the two copies of $\langle \star \rangle_i$ with the copies of some $\langle \oplus \rangle_i$ as in [Figure 3.9](#). By merging every other factor with its own copy, we effectively replace one $\langle \star \rangle_i$ with one $\langle \oplus \rangle_i$. We repeat this step until there are no $\langle \star \rangle_i$ left.

This step can be visualised as follows:



3. Finally, by copying every factor and then merging each with its own copy except for a number of $\langle \pm \rangle_i$, we can create any additional number of $\langle \pm \rangle_i$, until we have some $\hat{e} = \sqcup_{\hat{\theta}}(\hat{e}_1, \dots, \hat{e}_m)$ with $k \langle \pm \rangle_i$. Since every rewriting step preserves equivalence of the ω -closures and the fitting of the trajectories, it follows that $\hat{e}^\omega \equiv e^\omega$ and that $\hat{\theta}$ fits every \hat{e}_j .

This step can be visualised as follows:



□

Summarising, given $e_1 \cdot e_2^\omega$, by applying [Lemmas 3.22](#) and [3.23](#) we can rewrite e_1 as a shuffle of factors $\langle \pm \rangle_i$, $\langle \varepsilon \rangle_i$, $\langle + \rangle_i$, and e_2^ω as a disjunction of shuffles of factors $\langle \pm \rangle_i$, $\langle \varepsilon \rangle_i$, $\langle \pm \rangle_i^\omega$, $\langle \pm \rangle_i^\omega$, such that the number of $\langle \pm \rangle_i^\omega$ in every term of the disjunction equals the number of $\langle + \rangle_i$ in e_1 . By applying the laws of distributivity, we can then rewrite $e_1 \cdot e_2^\omega$ as a disjunction of concatenations of shuffles. Since the numbers of $\langle + \rangle_i$ and $\langle \pm \rangle_i^\omega$ match in every term of this disjunction, we can apply [Lemma 3.11](#) to merge every pair into $\langle \pm \rangle_i^\omega$. Since all factors are now balanced, every balanced ω -regular language has a corresponding expression in Ω^ω :

Theorem 3.24 (Completeness).

$$\{L(e) \mid e \in \Omega^\omega\} \supseteq \{L \mid L \text{ is a balanced } \omega\text{-regular language}\}.$$

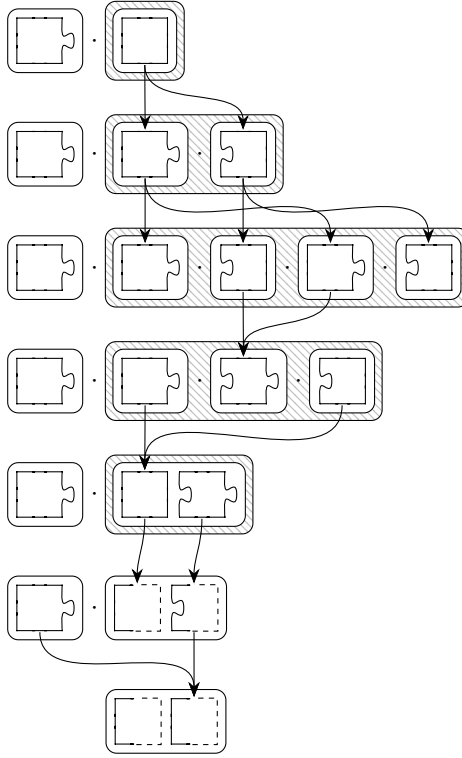


Figure 3.20: Construction of [Example 3.25](#) as groups of jigsaw pieces.

Example 3.25. We construct an expression in Ω^{\sqcup} for $\llbracket_1(\llbracket_1]_1)^\omega$.

$$\begin{aligned}
 \llbracket_1(\llbracket_1]_1)^\omega &\equiv \sqcup_1(\oplus_1^0)(\sqcup_{11}(\odot_1^1))^\omega \\
 &\equiv \sqcup_1(\oplus_1^0)(\sqcup_1(\oplus_1^0) \sqcup_1(\odot_1^0))^\omega \\
 &\equiv \sqcup_1(\oplus_1^0)(\sqcup_1(\oplus_1^0) \sqcup_1(\odot_1^0) \sqcup_1(\oplus_1^0) \sqcup_1(\odot_1^0))^\omega \\
 &\equiv \sqcup_1(\oplus_1^0)(\sqcup_1(\oplus_1^0) \sqcup_{11}(\oplus_1^0) \sqcup_1(\odot_1^0))^\omega \\
 &\equiv \sqcup_1(\oplus_1^0)(\sqcup_{1221}(\odot_1^1, \oplus_1^0))^\omega \\
 &\equiv \sqcup_1(\oplus_1^0) \sqcup_{(1221)^\omega}(\odot_1^\omega, \oplus_1^\omega) \\
 &\equiv \sqcup_{1(2112)^\omega}(\odot_1^\omega, \odot_1^\omega).
 \end{aligned}$$

Figure 3.20 illustrates this construction using groups of jigsaw pieces, line by line. The ω -operator is visualized as an extra hatched container.

3.6 Conclusion

As shown by Theorems 3.10, 3.13, 3.21 and 3.24, there exist grammars which can express precisely all balanced regular and ω -regular languages (Figures 3.5 and 3.19, respectively). When translated back to choreographies, this shows that it is possible to design choreography languages that can express all communication-error-free regular (and ω -regular) communication protocols. This answers Question 1 positively. Fundamentally, this chapter shows that the syntactic restrictions of choreography languages, i.e., using full communications rather than separate send and receive actions, are not necessarily a limiting factor in their expressiveness.

As a brief example, the behaviour depicted in Figure 1.7, which we could not specify using the operators shown in Chapter 1, could be specified using the shuffle on trajectories operator as $\sqcup_{(12+21)(12+21)}(C \rightarrow Y: \text{ball}, Y \rightarrow C: \text{ball})$.

We note that this chapter does not consider realisability, which is an important reason for limiting the expressiveness of choreography languages. Furthermore, the shuffle on trajectories operator is versatile, but it is not straightforward to use, whether in design or in analysis. The operator (when generalised to languages or expressions) matches words from its operands and filters out invalid combinations; it is easy to accidentally allow either too little or too much in a specification, and this subtle behaviour makes it non-trivial to, for example, define the projection of expressions containing shuffles on trajectories. We see more promise in using the operator in the theoretical foundations of choreography languages: since it generalises existing operators — notably sequential and parallel composition — it could be used to consolidate proofs, and make it easier to add other new operators that can be encoded using shuffles on trajectories.

We end with a discussion on two open problems related to this chapter: extending the results to context-free languages, and achieving the same results without resorting to the multiary version of the shuffle on trajectories operator.

Balanced context-free languages

Having dealt with balanced regular and ω -regular languages, the natural next step is to consider balanced context-free languages. As these are languages of finite words, we can reuse the definition of balancedness as in Section 3.2.

In the same way we characterised balanced regular languages in terms of syntactic constraints on finite automata and regular expressions, it is possible to characterise balanced context-free languages in terms of syntactic constraints on pushdown automata and context-free grammars. In both cases, we add a certificate ∇ , just as for finite automata. For pushdown automata, this certificate does not only consider the state but also the content of the stack; for context-free grammars, the certificate assigns a value to the grammar's variables and terminals.

However, developing a balanced-by-construction grammar for context-free languages is not as straightforward as adding the shuffle on trajectories operator to a context-free grammar, such as the regular-like expressions described by Gruska [Gru71]. The main complication is that context-free languages are not closed under the shuffle on trajectories operator, i.e., the shuffle of a set of context-free languages on a context-free language of trajectories is not necessarily context-free. In fact, Mateescu et al. show that $\sqcup_T(L_1, L_2)$ is guaranteed to be context-free only if at least two of T , L_1 and L_2 are regular and the third is context-free [MRS98].

This holds even for seemingly simple sets of trajectories. For example, the context-free language $[\binom{n}{1} [\binom{n}{2} \binom{n}{1}]_2]_2$ can be expressed as $\sqcup_{1^m 2^1 m^2}(\binom{n}{1} \binom{n}{1}, [\binom{n}{2}]_2)$. However, when we use the same context-free set of trajectories to shuffle the context-free language $L_1 = a^n b^n c^*$ with the regular language $L_2 = d^*$, the result is not context-free. To see this, note that $\sqcup_{1^m 2^1 m^2}(a^n b^n c^*, d^*) \cap a^+ b^+ d^+ c^+ d^+ = a^n b^n d c^{2n} d$, which is not context-free; as context-free languages are known to be closed under intersection with regular languages, $\sqcup_{1^m 2^1 m^2}(a^n b^n c^*, d^*)$ cannot be context-free.

Developing a balanced-by-construction grammar for context-free languages thus is a non-trivial open problem. We note that there exists further research on grammars with trajectories [Mar+99; OS05], which may inspire future efforts towards this goal.

The same investigation could be made for visibly pushdown languages, a subset of context-free languages with many desirable closure and decidability properties [AM04], and for deterministic context-free languages. For both of these, results may thus be easier to obtain.

Binary shuffles

Another non-trivial question is whether the grammar \mathbb{E}^\sqcup in Figure 3.5 is equally expressive if we restrict the shuffle on trajectories operator to its form with

two operands: is it possible to rewrite any n -ary shuffle to a combination of (nested) binary ones?

As a simple first example, take $\sqcup_{123123}([1]_1, [2]_2, [3]_3) \equiv [1][2][3]_1]_2]_3$. This ternary shuffle can straightforwardly be rewritten with nested binary shuffles as $\sqcup_{122122}([1]_1, \sqcup_{1212}([2]_2, [3]_3))$. The reason this example is easy, is because different types of parentheses can be neatly isolated into separate operands of the (nested) binary shuffles.

As a more complicated second example, take:

$$\sqcup_{1(2112)^*(3131)^*1}([1]_1)^*, ([2]_2)^*, ([3]_3)^* \equiv [1]([2]_1[1]_2)^*([3]_1]_3[1]_1)^*]_1$$

The first difficulty is that different types of parentheses cannot be neatly isolated as in the previous example. Moreover, a second difficulty is that the two loops in the trajectory expression mix the loops of the operands; as a result, neither one of those loops can be neatly isolated. To overcome these difficulties, a “trick” that we can use is to decompose the expression based on the number of times that the second loop (in the trajectory expression) is repeated: 0 times or at least once. In the former case, the expression simplifies to $[1]([2]_1[1]_2)^*]_1$; this one is straightforward to rewrite. In the latter case, the expression simplifies to $[1]([2]_1[1]_2)^*[3]_1]_3[1]([3]_1]_3[1]_1)^*]_1$, which can be broken down as the concatenation of $[1]([2]_1[1]_2)^*[3]_1]_3$ and $[1]([3]_1]_3[1]_1)^*]_1$, both of which are straightforward to write with binary shuffles by extracting single pairs of parentheses at a time.

We believe it is possible to prove that this “trick” can be generalised to deal with any number of concatenated loops, i.e., any expression $v_0w_1^*v_1w_2^*\dots w_n^*v_n$ where all v_i and w_i are words. Consequently, the generalised “trick” can deal with any balanced regular expression without nested loops.

However, the generalised “trick” does not seem to work for nested loops. As a counterexample, consider $[1]([2]_1([3]_3]_2[2]_2]_3]_3)^*[1]_2)^*]_1$. A straightforward attempt would be to isolate the inner loop (and the parentheses that occur in it): $\sqcup_{1(21(222222)^*12)^*1}([1]_1)^*, ([2]([3]_3]_2[2]_2]_3]_3)^*[1]_2)^*$. However, the shuffle has no way to distinguish, e.g., $[2]_2[2]_2[2]_2[2]_2$ (four iterations of the outer loop and none of the inner) and $[2]_2[3]_3]_2[2]_2[3]_3]_2$ (one iteration of both), as both have length 8. As a result, it also accepts the word $[1]([2]_1]_2[2]_2[2]_2[2]_2[1]_2)^*]_1$, which is not in the original language. We have as of yet found no way to avoid this ambiguity and thus to express this and expressions with nested loops in general using only binary shuffle on trajectories operators.

Branching pomsets

This chapter is based on section 3 of our ICE 2022 paper [ICE22] and on sections 2 and 3 of the subsequent JLAMP 2024 paper [JLAMP24]. It also contains part of the conclusions and related work of these papers. It partially answers [Question 2](#).

In this chapter, we formally define the syntax and semantics of branching pomsets (BPs), as described in [Chapter 1](#), and we compare their expressiveness with that of several classes of event structures.

4.1 Syntax

The general idea of a branching pomset is that it represents all possible events, but some of them are selected through a choice. We denote choices visually as choice boxes containing branches, e.g., in [Figures 1.10](#) and [4.1](#). The branching structure does not interrupt the causality relation and all events still participate in it, as shown in the examples, where arrows point both into and out of the branches of the choice. Nested choices are supported as well.

Formally, the branching structure of a BP is a tree whose leaves are events and whose inner nodes represent a structure of (possibly nested) choices and branches. It is defined below with root node \mathcal{B} , whose children \mathcal{C} are either a single event e (from the BP) or a binary choice node with children (branches)

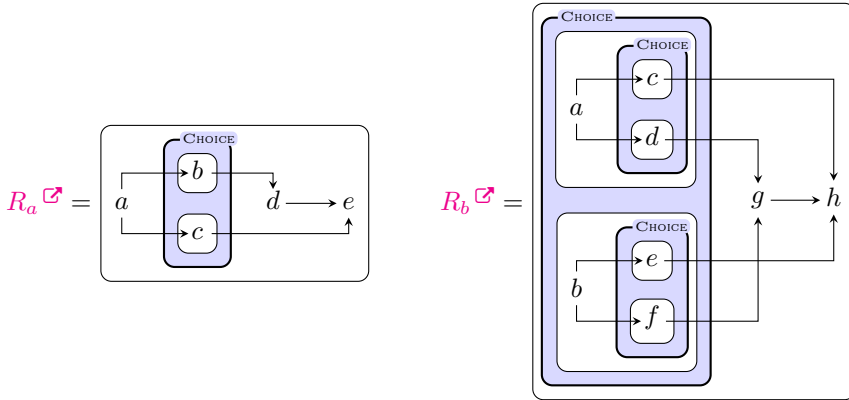


Figure 4.1: Two branching pomsets.

$\mathcal{B}_1, \mathcal{B}_2$. Note that a node \mathcal{C} must either be a leaf or an inner node with exactly two children. A node \mathcal{B} can have arbitrarily many children and can thus also be empty — including the root node.

Definition 4.1 (Branching structure).

$$\begin{aligned} \mathcal{B} &::= \{\mathcal{C}_1, \dots, \mathcal{C}_n\} \quad (n \geq 0) \\ \mathcal{C} &::= e \mid \{\mathcal{B}_1, \mathcal{B}_2\} \end{aligned}$$

Example 4.2. For the BP R_b in Figure 4.1, the set of events is $\{a, b, c, d, e, f, g, h\}$ and the labelling function maps each event to itself. The novelty with respect to pomsets is the branching structure $\mathcal{B} = \{g, h, \mathcal{C}_1\}$, where $\mathcal{C}_1 = \{\{a, \mathcal{C}_2\}, \{b, \mathcal{C}_3\}\}$, $\mathcal{C}_2 = \{\{c\}, \{d\}\}$ and $\mathcal{C}_3 = \{\{e\}, \{f\}\}$ are the outer and the two inner choices respectively.

We write $\mathcal{B}_1 \succeq \mathcal{B}_2$ if \mathcal{B}_1 is a subtree of \mathcal{B}_2 and $\mathcal{B}_1 \succ \mathcal{B}_2$ if \mathcal{B}_1 is a strict subtree of \mathcal{B}_2 , i.e., if $\mathcal{B}_1 \succeq \mathcal{B}_2$ and $\mathcal{B}_1 \neq \mathcal{B}_2$.¹² We use the same notation for nodes \mathcal{C} , events e (a special case of \mathcal{C}) and combinations of all the aforementioned. Formally, the subtree relation is defined below, where \mathcal{D} can be either a \mathcal{B} or a \mathcal{C} (and thus also a singleton e).

¹²We use a different notation than in the original papers and their technical reports, where we used \preceq and \prec for the subtree relation, to be more consistent with existing notation for event structures.

Definition 4.3 (Subtree).

$$\frac{}{\mathcal{D} \succeq \mathcal{D}} \quad \frac{\mathcal{D}_1 \in \mathcal{D}_2}{\mathcal{D}_1 \succeq \mathcal{D}_2} \quad \frac{\mathcal{D}_1 \succeq \mathcal{D}_2 \quad \mathcal{D}_2 \succeq \mathcal{D}_3}{\mathcal{D}_1 \succeq \mathcal{D}_3} \quad \frac{\mathcal{D}_1 \succeq \mathcal{D}_2 \quad \mathcal{D}_1 \neq \mathcal{D}_2}{\mathcal{D}_1 \succ \mathcal{D}_2}$$

Branching pomsets themselves are then formally defined below.

Definition 4.4 (Branching pomset). A branching pomset (BP) is a four-tuple $R = (E, \preceq, \lambda, \mathcal{B})$, where E is a set of events, $\preceq \subseteq E \times E$ is the causality relation such that \preceq^* (the reflexive and transitive closure of \preceq) is a partial order on events¹³, $\lambda : E \rightarrow \mathcal{L}$ is a labelling function assigning to every event a label in a labelling set \mathcal{L} , and \mathcal{B} is a branching structure such that the set of leaves of \mathcal{B} is E and no event in E occurs in \mathcal{B} more than once.

Formally, as an extension of pomsets, BPs should be defined as isomorphic classes, which allows us to abstract away from the names of events. To simplify matters, we have not done so here; it may thus be more accurate to consider them branching lposets. Furthermore, we note that, in contrast to traditional pomsets, the causality relation \preceq is not necessarily a partial order. We briefly discuss this in [Section 4.4](#). We use $R.E$, $R.\preceq$, $R.\lambda$ and $R.\mathcal{B}$ to refer to the components of R . We generally omit the prefix if doing so causes no confusion. We also write $e_1 < e_2$ if $e_1 \preceq e_2$ and $e_1 \neq e_2$. We use the following terminology:

- An event e is *minimal* (in R) if $e' \not\prec e$ for all $e' \in R.E$ (i.e., there exists no other event e' that precedes e).
- An event e is *active* (in R) if $e \in R.\mathcal{B}$ (i.e., e is not part of a choice).
- An event e is *enabled* (in R) if it is both active and minimal; we write $en(R)$ to denote the set of enabled events in R .
- Two events e_1 and e_2 are *causally ordered* (in R) if either $e_1 \preceq e_2$ or $e_2 \preceq e_1$.
- Two events e_1 and e_2 are *mutually exclusive* (in R) if there exists some $C = \{\mathcal{B}_1, \mathcal{B}_2\} \succ R.\mathcal{B}$ such that $e_1 \succ \mathcal{B}_1$ and $e_2 \succ \mathcal{B}_2$.

Remark. Technically, there is nothing prohibiting a pair of events from being both causally ordered and mutually exclusive: they may be part of different branches of the same choice, while one is dependent on the other. However,

¹³We use a different notation than in the original papers and their technical reports, where we used \leq for the causality relation. This is to make clearer that \preceq is not necessarily a partial order.

any such dependency is redundant and will not change the semantics of the BP. We thus assume that these two relations are disjoint.

4.2 Semantics

Informally, every execution step of a BP R , in which an event e is fired, is brought about in three steps:

1. First, R is optionally *refined* to a “sub-BP” R' by resolving zero (i.e., $R = R'$), one, or more choices. Each resolution is done by replacing a choice $\{\{\mathcal{B}_1, \mathcal{B}_2\}\}$ at any level of the branching structure with one of its branches \mathcal{B}_i , thereby discarding the other branch \mathcal{B}_j . We note that this same idea governs the operational semantics of many existing languages, too. For instance, in process calculi, if P can reduce to P' , then also $P + Q$ (i.e., free choice between P and Q) can reduce to P' , thus resolving the choice and discarding Q .
2. Second, an *enabling* is sought. If an event e in R' is enabled and, additionally, e is disabled in every refinement R'' of R that is larger than R' (i.e., fewer choices are resolved in R'' than in R'), then refining R to R' is said to be an *enabling* of e . In other words, R' is the largest sub-BP of R in which e is enabled (i.e., the smallest number of choices are resolved to enable e). If $R = R'$, then zero choices were resolved in the first refinement step. In contrast, if $R \neq R'$, then one or more choices were resolved to enable e : either because e was a minimal event of a branch in R that was chosen in R' (so e also became active), or because e was active in R and causally ordered after events in a branch in R that was discarded in R' (so e also became minimal) — or both.

Intuitively, a BP can be considered to represent a set of pomsets. Each resolution of a choice in a BP discards a number of these pomsets. Selecting the largest refinement as described above then corresponds to selecting the subset of all pomsets in which the chosen event is minimal.

3. Third, R is *reduced* by firing e . The resulting BP is R' (the chosen refinement of R) without e .

The empty BP cannot perform execution steps; it is said to *terminate*. Any BP which can refine to the empty BP is able to terminate.

Formally, execution and termination are defined through a number of relations.

$$\frac{}{\mathcal{B} \sqsupseteq \mathcal{B}} [\text{REFL}] \quad \frac{\mathcal{B} \sqsupseteq \mathcal{B}' \sqsupseteq \mathcal{B}''}{\mathcal{B} \sqsupseteq \mathcal{B}''} [\text{TRANS}] \quad \frac{i \in \{1, 2\} \quad \{\mathcal{B}_1, \mathcal{B}_2\} \notin \mathcal{B}}{\{\{\mathcal{B}_1, \mathcal{B}_2\}\} \cup \mathcal{B} \sqsupseteq \mathcal{B}_i \cup \mathcal{B}} [\text{CHOICE}]$$

$$\frac{\mathcal{B}_1 \sqsupseteq \mathcal{B}'_1 \quad \mathcal{B}_2 \sqsupseteq \mathcal{B}'_2 \quad \{\mathcal{B}_1, \mathcal{B}_2\} \notin \mathcal{B}}{\{\{\mathcal{B}_1, \mathcal{B}_2\}\} \cup \mathcal{B} \sqsupseteq \{\{\mathcal{B}'_1, \mathcal{B}'_2\}\} \cup \mathcal{B}} [\text{CONGR}] \quad \frac{R.\mathcal{B} \sqsupseteq \mathcal{B}'}{R \sqsupseteq R|_{\mathcal{B}'}} [\text{LIFT}]$$

(a) Refinement rules.

$$\frac{R \sqsupseteq R' \quad e \in \text{en}(R') \quad \forall R'' : R \sqsupseteq R'' \sqsupseteq R' \Rightarrow e \notin \text{en}(R'')}{R \xrightarrow{e} R'} [\text{ENABLE}]$$

$$\frac{R \xrightarrow{e} R'}{R \xrightarrow{e} R' - e} [\text{REDUCE1}] \quad \frac{R \xrightarrow{e} R'}{R \xrightarrow{\lambda(e)} R'} [\text{REDUCE2}] \quad \frac{R.\mathcal{B} \sqsupseteq \emptyset}{R \downarrow} [\text{TERMINATE}]$$

(b) Enabling, reduction, and termination rules.

$$(E, \preceq, \lambda, \mathcal{B})|_{\mathcal{B}'} = (E', \preceq \cap (E' \times E'), \lambda \cap (E' \times \mathcal{L}), \mathcal{B}'),$$

where $E' = \{e \mid e \succ \mathcal{B}'\}$

$$\text{en}(R) = \{e \in R.E \mid e \in R.\mathcal{B} \wedge \nexists e' \in R.E : e' \prec e\}$$

$$\hat{e} - e = \hat{e}$$

$$\{\mathcal{C}_1, \dots, \mathcal{C}_n\} - e = \begin{cases} \{\mathcal{C}_1, \dots, \mathcal{C}_{i-1}, \mathcal{C}_{i+1}, \dots, \mathcal{C}_n\} & \text{if } \mathcal{C}_i = e \\ \{\mathcal{C}_1 - e, \dots, \mathcal{C}_n - e\} & \text{otherwise} \end{cases}$$

$$\{\mathcal{B}_1, \mathcal{B}_2\} - e = \{\mathcal{B}_1 - e, \mathcal{B}_2 - e\}$$

$$R - e = R|_{R.\mathcal{B} - e}$$

(c) Operations on branching pomsets.

Figure 4.2: Semantics of branching pomsets.

- **Refinement:** A branching structure \mathcal{B} can refine to \mathcal{B}' , written $\mathcal{B} \sqsupseteq \mathcal{B}'$. We write $\mathcal{B} \sqsupsetneq \mathcal{B}'$ to specify that $\mathcal{B} \neq \mathcal{B}'$. The refinement rules are formalised in Figure 4.2a. To illustrate these, we use BPs R_a, R_b in Figure 4.1. As the labels are irrelevant for these examples, we use a, \dots, h for both the events and their labels. Finally, we assume that the relation

- **Enabling, reduction, and termination:** Figure 4.2b defines an enabling relation, two reduction relations, and a termination predicate.

- The first rule, `ENABLE`, defines the conditions for enabling an event e , written $R \xrightarrow{\check{e}} R'$: a BP R can enable e by refining to R' if e is enabled in R' ($e \in en(R')$), and if there is no other refinement R'' in between which already enables e .

Example 4.5. Let $R_a, R_b, R'_a, R'_b, R''_b$ be as in Figures 4.1 and 4.3. Then:

$$* R_a \xrightarrow{\check{d}} R'_a$$

$$* R_b \xrightarrow{\check{a}} R'_b$$

$$* R_b \xrightarrow{\check{g}} R''_b$$

As in the informal description, the enabling relation only discards the absolutely necessary: for example, in the BP R_a in Figure 4.1, we may discard the choice's upper branch to fire d , but not to fire a . Similarly, in the BP R_b in the same figure, we may discard the lower branch from both inner choices to fire g , but there is no need to also resolve the outer choice.

The refinement rules in Figure 4.2a act as structural rules, which do not fire any event but may exclude events (by discarding branches), as opposed to the reduction rules in Figure 4.2b, which fire events and are therefore computational rules. In fact, refinements could also be seen as executing silent transitions to resolve choices, as in process algebras with an internal choice operator.

- The second and third rules, `REDUCE1` and `REDUCE2`, define the reduction relations. They state that, if R can enable e by refining to R' (through `ENABLE`), then it can fire e by reducing to $R' - e$, which is the BP obtained by removing e from R' (Figure 4.2c). This reduction is defined both on e 's label (`REDUCE2`) and on the event itself (`REDUCE1`), the latter for internal use in proofs since $\lambda(e)$ is typically not unique but e is.
- The fourth rule, `TERMINATE`, defines the termination predicate and simply states that a BP can terminate if its branching structure can reduce to the empty set.

4.3 Comparison with event structures

In this section, we study the expressive power of branching pomsets by comparing them with various classes of event structures.

Event structures (ESs) are a well-established model for concurrency which bear a close relationship with both Petri nets and domain theory. Originally introduced by Nielsen, Plotkin and Winskel [Win80; NPW81], this model represents a concurrent system as a set of (possibly labelled) events together with some relations among them, which regulate their occurrence in computations. Typically, in a prime event structure (PES), the original and simplest form of ES, there are two relations on events: *causality* (meaning that one event must occur before the other), and *conflict* (meaning that two events cannot occur in the same computation).¹⁵

In their labelled version, PESs may be viewed as pomsets enriched with a conflict relation. This makes them conceptually very close to branching pomsets, therefore a comparison is in order.

Many variants of ESs have been studied in the last decades. We start by reviewing a number of them, referring for more details to the extensive overview given in the paper by Arbach et al. [Arb+18].

4.3.1 Event structure landscape

We first introduce the classes of ESs that are relevant to our study, namely those represented in Figure 4.4 (except for the faded ones, which are only included for completeness). We then uniformly define their semantics using the notion of *proving sequence*, from which the classical notion of *configuration* (a set of events that may have occurred at some stage of computation) may be immediately derived.

We start by considering the classes of *static* ESs, namely prime, bundle, asymmetric, and extended bundle ESs, where the relations on events are fixed once and for all. We then move to the classes of *dynamic* ESs, namely growing, shrinking and dynamic causality ESs, as well as ESs for resolvable conflict, where one of the two relations of causality and conflict may vary along execution.

¹⁵To be more precise, the simplest class of ESs is that of *elementary* ESs, where the conflict relation is empty [NPW81]. Elementary ESs are obtained from *causal nets* by retaining only their events and dropping their conditions. In fact, elementary ESs are just posets of events, or, in their labelled version, pomsets of event labels.

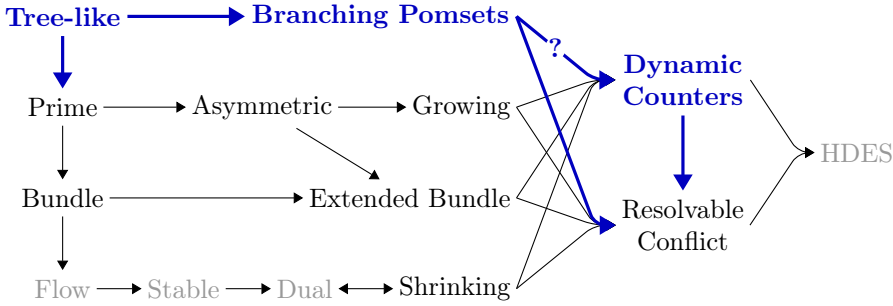


Figure 4.4: Landscape of event structures as extended from [Arb+18]. Branching pomsets are added in bold. Dynamic causality event structures are replaced by a variant with counters. Flow, stable, dual, and higher order dynamic causality event structures are faded to indicate that they are not discussed in detail.

Static event structures

The simplest class of ESs we consider, which is also the original one, is that of prime ESs.

Definition 4.6 (Prime Event Structure (PES) [Win80]). A prime event structure is a triple $S = (E, \#, \leq)$, where E is a set of events, $\# \subseteq E \times E$ is a symmetric, irreflexive relation called the *conflict* relation, and $\leq \subseteq E \times E$ is a partial order relation called the *causality* relation, satisfying the properties:

$$[e] = \{e' \mid e' \leq e\} \text{ is finite for all } e, e' \in E \quad (\text{finite causes})$$

$$e \# e' \wedge e' \leq e'' \implies e \# e'' \quad (\text{conflict hereditariness})$$

The condition of **conflict hereditariness** implies that the relations of conflict and causality are disjoint and that events do not have conflicting causes. Two events which are neither in conflict nor causally related are said to be *concurrent*. Two events which are not in conflict are said to be *consistent*. The axiom of **finite causes** forbids events with an infinite set of (consistent) causes, namely an event e such that $e_n \leq e$ for all $n \in \mathbb{N}^+$, thus also ruling out infinite regression (which could arise if we also had $e_{n+1} \leq e_n$ for all $n \in \mathbb{N}^+$).

Event structures were originally conceived as a system model and not as an algebraic model endowed with a set of constructors. However, since ESs were introduced roughly at the same time as the first process calculi CSP, CCS

and ACP, they appeared as a natural candidate model¹⁶ to give semantics to process calculi. Because of their inability to represent disjunctive causality (i.e., events with multiple possible causes, only one of which needs to happen), PESs turned out to be too restrictive for that purpose. Typically, when two CCS processes are composed in parallel, some of their events (those carrying complementary labels) are allowed to synchronise giving rise to a new event, whose set of successor events should be the union of the sets of successors of the two original events, which can still occur independently. However, in a PES the conflict hereditariness condition prevents any sharing of the sets of successors, thus requiring their duplication after each synchronisation.

To overcome this problem, Winskel devised a more general class of ESs, called *stable event structures* [Win80], which accommodates disjunctive causality by replacing the causality relation \leq with an *enabling* relation between sets of events and events. Hence, the first ES semantics for CCS [Win82] was given in terms of stable ESs. However, since the enabling relation is a second-order relation on events, stable ESs lose the nice graphical representation offered by PESs. This observation motivated the subsequent introduction of two subclasses of stable ESs, respectively bundle ESs [Lan93] and flow ESs [BC88], which allow for disjunctive causality while retaining a graphical representation. Both these classes of ESs may be viewed as simple extensions of PESs, flow ESs being slightly more expressive than bundle ESs [BC91]. Here we only consider bundle ESs, which use a simpler enabling relation than stable ESs.

Definition 4.7 (Bundle Event Structure (BES) [Lan93]). A bundle event structure is a triple $S = (E, \#, \rightsquigarrow)$, where E is a set of events, $\# \subseteq E \times E$ is the symmetric, irreflexive conflict relation and $\rightsquigarrow \subseteq \mathcal{P}(E) \times E$ is the enabling relation, satisfying the property:

$$X \rightsquigarrow e \implies \forall e_1 \neq e_2 \in X : e_1 \# e_2 \quad (\text{stability})$$

Intuitively, $X \rightsquigarrow e$ means that *at least* one of the events in X needs to happen before e can happen. The condition of **stability** furthermore implies that *at most* one of the events in X needs to happen, as all the events in X must be pairwise in conflict with each other.

For both PESs and BESs, more general variants where conflict is not required to be symmetric have been proposed. These are called respectively asymmetric ESs and extended bundle ES. We recall their definitions.

¹⁶in their labelled version, where events have labels that represent process actions or communications.

Definition 4.8 (Asymmetric Event Structure (AES) [BCM01]). An asymmetric event structure is a triple $S = (E, \rightsquigarrow, \leq)$, where E is a set of events, $\rightsquigarrow \subseteq E \times E$ is the *asymmetric conflict* relation, and $\leq \subseteq E \times E$ is the partially ordered *causality* relation, satisfying the following properties for all $e, e', e'' \in E$:

$$[e] = \{e' \mid e' \leq e\} \text{ is finite} \quad (\text{finite causes})$$

$$e < e' \implies e \rightsquigarrow e' \quad (4.1)$$

$$(e \rightsquigarrow e' \wedge e' < e'') \implies e \rightsquigarrow e'' \quad (4.2)$$

$$\rightsquigarrow \text{ is acyclic on } [e] \quad (4.3)$$

$$(\rightsquigarrow \text{ cyclic on } [e] \cup [e']) \implies e \rightsquigarrow e' \quad (4.4)$$

In the above definition, Condition 4.2 expresses hereditariness of asymmetric conflict, while Condition 4.3 rules out cycles of asymmetric conflict in the set of causes of an event. Since cycles can be self-cycles, this implies in particular that asymmetric conflict is irreflexive. As for Condition 4.4, it requires that any semantic conflict between two events which is due to their co-occurrence on a cycle of asymmetric conflict be explicitly represented by an asymmetric conflict in both directions.

As explained in [BCM01], the asymmetric conflict relation has two natural interpretations: $e \rightsquigarrow e'$ may be understood as (i) e' *disables* e , namely the occurrence of e' prevents the occurrence of e , or (ii) e (*weakly*) *precedes* e' , namely e occurs before e' in all executions where they both occur.

A similar disabling relation is used in extended bundle ESs.

Definition 4.9 (Extended Bundle Event Structure (EBES) [Lan92]). An extended bundle event structure is a triple $S = (E, \rightsquigarrow, \succ)$, where E is a set of events, $\rightsquigarrow \subseteq E \times E$ is the irreflexive *disabling* relation and $\succ \subseteq \mathcal{P}(E) \times E$ is the *enabling* relation, satisfying the following:

$$X \succ e \implies \forall e_1 \neq e_2 \in X : e_1 \rightsquigarrow e_2 \quad (\text{stability condition})$$

Here again, $e \rightsquigarrow e'$ should be read from right to left as e' *disables* e .

The construction of an EBES from an AES is similar to that of a BES from a PES. As such, AESs are included in EBESs.

Dynamic event structures

We now review the classes of dynamic ESs, where one of the two relations of causality and conflict may dynamically change along execution.

In [Arb+18], three new classes of ESs have been proposed, where the causality relation can be modified by effect of the occurrence of some other event: (1) *shrinking causality event structures (SESSs)*, where causal dependencies can be removed, (2) *growing causality event structures (GESs)*, where causal dependencies can be added, and (3) *dynamic causality event structures (DCESSs)*, where causal dependencies can be both added and removed.

Shrinking causality ESs extend rPESs (relaxed PESs where the conflict relation is not required to be hereditary¹⁷) by allowing the causality relation to decrease along execution.

Definition 4.10 (Shrinking Causality Event Structure (SES) [Arb+18]). A shrinking causality event structure is a quadruple $S = (E, \#, \rightarrow, \triangleleft)$, where E is a set of events, $\# \subseteq E \times E$ is the symmetric, irreflexive *conflict* relation, $\rightarrow \subseteq E \times E$ is the *initial causality* relation and $\triangleleft \subseteq E \times E \times E$ is the *shrinking causality* relation (written $e \triangleleft [e_1 \rightarrow e_2]$ for $(e, e_1, e_2) \in \triangleleft$) satisfying the property:

$$e \triangleleft [e_1 \rightarrow e_2] \implies (e_1 \rightarrow e_2 \wedge e \notin \{e_1, e_2\}) \quad (\text{SC})$$

Note that SESs can model disjunctive causality. Indeed, the shrinking causality $e \triangleleft [e' \rightarrow e'']$ represents a situation where initially e' causes e'' , but this causality may be cancelled by the occurrence of e . Thus, if $e \triangleleft [e' \rightarrow e'']$ and $e' \triangleleft [e \rightarrow e'']$ and $e \# e'$, then e and e' are two conflicting causes of e'' . On the other hand, if $e \triangleleft [e' \rightarrow e'']$ and $e' \triangleleft [e \rightarrow e'']$ and $\neg(e \# e')$, then both e and e' may occur in the same computation, thus we can reach a state where all of e, e', e'' have occurred: in this state, we do not know which of e or e' has caused e'' (one of them must, since e'' cannot occur alone). This means that SESs are not stable. Moreover, SESs cannot model disabling: since causalities can only be dropped and never added, an event that becomes at any point will always remain so.

Dually, growing causality ESs extend rPESs by allowing the causality relation to increase along execution.

¹⁷While the name ‘‘rPES’’ has been coined in [Arb+18], this PES variant had already been used to interpret a subset of CCS in [BC87].

Definition 4.11 (Growing Causality Event Structure (GES) [Arb+18]). A growing causality event structure is a triple $S = (E, \rightarrow, \blacktriangleright)$, where E is a set of events, $\rightarrow \subseteq E \times E$ is the *initial causality* relation and $\blacktriangleright \subseteq E \times E \times E$ is the *growing causality* relation (written $e \blacktriangleright [e_1 \rightarrow e_2]$ for $(e, e_1, e_2) \in \blacktriangleright$) satisfying the property:

$$e \blacktriangleright [e_1 \rightarrow e_2] \implies (\neg(e_1 \rightarrow e_2) \wedge e \notin \{e_1, e_2\}) \quad (\text{GC})$$

Note that conflict does not appear in Definition 4.11. This is because conflict may be simulated by mutual disabling, and disabling may be simulated by growing causality. The simulation of disabling as given in [Arb+18] assumes the existence of an “impossible” event e_{imp} ¹⁸ such that $e_{imp} \rightarrow e_{imp}$ (exploiting the fact that \rightarrow is not required to be irreflexive): then a conflict between e and e' may be modelled by setting $e \blacktriangleright [e_{imp} \rightarrow e']$ and $e' \blacktriangleright [e_{imp} \rightarrow e]$.

In fact, since Definition 4.11 does not require $e_1 \neq e_2$ in Condition (GC), the conflict between e and e' may also be simulated more directly (see [Arb+18] again) by letting $e \blacktriangleright [e' \rightarrow e']$ and $e' \blacktriangleright [e \rightarrow e]$. Note finally that, unlike SESs, GESs cannot model disjunctive causality.

Combining the features of shrinking and growing ESs, and adding some constraints on the interplay between shrinking and growing causality, we obtain dynamic causality ESs (DCEs). In this chapter, we consider a variant of DCEs, with the same syntax but subtly different semantics. This is discussed when we formally define the semantics.

Definition 4.12 (Dynamic Causality Event Structure [Arb+18]). A dynamic causality event structure (DCES) is a quadruple $S = (E, \rightarrow, \triangleleft, \blacktriangleright)$, where E is a set of events, $\rightarrow \subseteq E \times E$ is the *initial causality* relation, and $\triangleleft, \blacktriangleright \subseteq E \times E \times E$ are respectively the *shrinking* and *growing causality* relations, such that:

$$\nexists e' \in E : e' \blacktriangleright [e_1 \rightarrow e_2] \wedge e \triangleleft [e_1 \rightarrow e_2] \implies e_1 \rightarrow e_2 \quad (4.5)$$

$$e \triangleleft [e_1 \rightarrow e_2] \implies e \notin \{e_1, e_2\} \quad (4.6)$$

¹⁸The existence of impossible events, namely events that cannot occur in any computation, is a common feature of all classes of ESs except PESs. When a particular ES does not contain such events, it is said to be *full*.

$$\nexists e' \in E : e' \triangleleft [e_1 \rightarrow e_2] \wedge e \blacktriangleright [e_1 \rightarrow e_2] \implies \neg(e_1 \rightarrow e_2) \quad (4.7)$$

$$e \blacktriangleright [e_1 \rightarrow e_2] \implies e \notin \{e_1, e_2\} \quad (4.8)$$

$$e \blacktriangleright [e_1 \rightarrow e_2] \implies \neg(e \triangleleft [e_1 \rightarrow e_2]) \quad (4.9)$$

Again, conflict is omitted because it may be simulated by growing causality. Conditions 4.6 and 4.8 correspond to the second half of Conditions (SC) and (GC) respectively. Conditions 4.5 and 4.7 correspond to the first half of Conditions (SC) and (GC), accounting for the possibility that the other operator could add or subtract some causal dependencies. Finally, Condition 4.9 prevents an event from adding and dropping the same causal dependency.

The last class of dynamic ESs we shall consider is the following, where the conflict relation may be changed dynamically.

Definition 4.13 (Resolvable Conflict Event Structure (RCES) [GP04]). An event structure for *resolvable conflict* is a pair $S = (E, \vdash)$, where E is a set of events and $\vdash \subseteq \mathcal{P}(E) \times \mathcal{P}(E)$ is the *enabling* relation.

We recall from [GP04] a simple example showing that RCESs can model resolvable conflicts, namely conflicts that disappear by effect of the occurrence of some event. Consider the structure (E, \vdash) where $E = \{a, b, c\}$, with $\{a\} \vdash \{b, c\}$ and $\emptyset \vdash X$ iff $X \subseteq E$ and $X \neq \{b, c\}$. This models an initial conflict between b and c , which can be resolved by the occurrence of a . We refer the reader to [GP04] for more examples, showing that RCESs are not stable. In fact, in [GP04] RCESs are shown to be able to represent any Petri net, a very strong expressiveness result.

Event structure semantics

The semantics of ESs is classically defined in terms of configurations. A *configuration* is a set of events that may have occurred at some stage of a computation. For all classes of ESs, we shall uniformly define a configuration to be a set of events enumerable as a *proving sequence* [Win80; BC91; Lan92], namely a sequence of consistent events such that each event is “secured” - i.e., granted the possibility to occur - by the preceding ones. We will first introduce proving sequences for all classes of ESs, and then uniformly derive from them a notion of configuration and a reduction relation on configurations.

We start by introducing some terminology. A *trace* is a finite sequence of distinct events $t = e_1 \dots e_n$, where $n \geq 0$ and by convention $t = \varepsilon$ if $n = 0$.

Given a trace $t = e_1 \dots e_n$, we denote by $t_i = e_1 \dots e_i$ its prefix of length i for every $i \leq n$, and by \bar{t} the set $\{e_1, \dots, e_n\}$ of events occurring in t . In particular, $t_n = t$, $t_0 = \varepsilon$ and $\bar{t}_0 = \emptyset$. As for pairs of events, a set of events is *consistent* if it is conflict-free.

Proving sequences are traces with two distinguishing properties: *consistency* of the underlying set of events, and *securing* for each of their events. Their formal definitions differ depending on the considered class of ESs.

Definition 4.14 (PES proving sequence [Win80]). Let $S = (E, \#, \leq)$ be a prime event structure. A proving sequence of S is a trace $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$, satisfying the properties:

$$\forall i, j \in [1, n] : \neg(e_i \# e_j) \quad (\text{consistency})$$

$$\forall i \in [1, n] : \forall e \in E : (e < e_i \implies e \in \overline{t_{i-1}}) \quad (\text{left-closure})$$

Definition 4.15 (AES proving sequence [BCM01]). Let $S = (E, \rightsquigarrow, \leq)$ be an asymmetric event structure. A proving sequence of S is a trace $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$, satisfying the properties:

$$\forall i, j \in [1, n] : e_i \rightsquigarrow e_j \implies i < j \quad (\text{consistency})$$

$$\forall i \in [1, n] : \forall e \in E : (e < e_i \implies e \in \overline{t_{i-1}}) \quad (\text{left-closure})$$

For PESs and asymmetric ESs, which have a global causality relation \leq , securing amounts to *left-closure* with respect to \leq . For more expressive ESs such as BESs and EBESs, which allow for disjunctive causality and only recover a partial order of causality within individual computations, securing is defined as *consistent left-closure*, namely left-closure up to conflicts. In BESs and EBESs, this property is called *bundle satisfaction*.

Definition 4.16 (BES proving sequence [Lan92]). Let $S = (E, \#, \rightsquigarrow)$ be a bundle event structure. A proving sequence of S is a trace $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$, satisfying the properties:

$$\forall i, j \in [1, n] : \neg(e_i \# e_j) \quad (\text{consistency})$$

$$\forall i \in [1, n] : \forall X \subseteq E : (X \rightsquigarrow e_i \implies X \cap \overline{t_{i-1}} \neq \emptyset) \quad (\text{bundle satisfaction})$$

Definition 4.17 (EBES proving sequence [Lan92]). Let $S = (E, \rightsquigarrow, \succ)$ be an extended bundle event structure. A proving sequence of S is a trace $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$ satisfying the properties:

$$\forall i, j \in [1, n] : e_i \rightsquigarrow e_j \implies i < j \quad (\text{consistency})$$

$$\forall i \in [1, n] : \forall X \subseteq E : (X \succ e_i \implies X \cap \overline{t_{i-1}} \neq \emptyset) \quad (\text{bundle satisfaction})$$

For dynamic classes of ESs, the definition of proving sequence is more subtle, since it needs to account for the fact that the causes of every event in the sequence may have been modified by some earlier event in the sequence. The definitions we give below for SESs and GESs are slightly different in form, but semantically equivalent, to the ones given in [Arb+18].

Definition 4.18 (SES proving sequence [Arb+18]). Let $S = (E, \#, \rightarrow, \triangleleft)$ be a shrinking causality event structure. A proving sequence of S is a trace $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$ satisfying the properties:

$$\forall i, j \in [1, n] : \neg(e_i \# e_j) \quad (\text{consistency})$$

$$\forall i \in [1, n] : \forall e \in E : \\ e \rightarrow e_i \implies (e \in \overline{t_{i-1}} \vee \exists e' \in \overline{t_{i-1}} : e' \triangleleft [e \rightarrow e_i]) \quad (\text{securing})$$

Informally, the securing property for SESs means that if e causes e_i initially ($e \rightarrow e_i$), then either e has indeed happened before e_i ($e \in \overline{t_{i-1}}$), or another event e' has happened that removed the causality ($\exists e' \in \overline{t_{i-1}} : e' \triangleleft [e \rightarrow e_i]$).

Definition 4.19 (GES proving sequence [Arb+18]). Let $S = (E, \rightarrow, \blacktriangleright)$ be a growing causality event structure. A proving sequence of S is a trace $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$ satisfying the property:

$$\forall i \in [1, n] : \forall e \in E : (e \rightarrow e_i \vee \exists e' \in \overline{t_{i-1}} : e' \blacktriangleright [e \rightarrow e_i]) \implies e \in \overline{t_{i-1}}$$

Informally, the securing property for GESs means that if either e causes e_i initially ($e \rightarrow e_i$), or if another event e_j has happened that added the causality ($e' \blacktriangleright [e \rightarrow e_i]$), then e has indeed happened before e_i ($e \in \overline{t_{i-1}}$).

Recall that DCESs combine the power of SESs and GESs: they are essentially PESs whose causality relations can both shrink and grow as events happen in a computation. Relative to SESs and GESs, the key complication to define the semantics of DCESs is that *the same causality* can be removed, added,

removed again, etc., in the same computation. In contrast, in SESs (resp. GESs), once a causality is removed (resp. added), it remains absent (resp. present) forever. Thus, a more advanced bookkeeping mechanism is needed to account for the additions/removals of causalities to define proving sequences of DCEs.

In [Arb+18] the operations for adding and dropping a cause are idempotent, in the sense that the executions of two successive additions (resp. removals) of the same causal dependency have the same effect as that of a single one. Here, we adopt a finer mechanism, which counts the number of additions and removals along an execution. Thus, our semantics for DCEs is a variant of the original one proposed in [Arb+18], which we will call *dynamic causality event structures with counters* (DCCESs). We provide a comparison of the original definition of DCEs with DCCESs and BPs in Appendix B. Specifically, we show that they are incomparable with both.

One advantage of this new semantics is that it supports a simple definition of the transition relation on configurations, which does not require the pairing of configurations with an ordering as in [Arb+18]. More precisely, the intuition for a trace $t = e_1 \dots e_n$ to be a proving sequence of a DCCES is that for any event e_i in the trace, any cause of e_i which has been added more times than it has been dropped by events occurring in the prefix t_{i-1} does effectively cause e_i at this stage of computation, and therefore it should occur in the prefix t_{i-1} . The definition relies on some auxiliary multisets, which are variations (enriched with multiplicities) of the auxiliary sets used in [Arb+18]. Clearly DCCESs extend both SESs and GESs. The proof for EBESs given in [Arb+18] also holds for DCCESs, thus giving the inclusions shown in Figure 4.4.

Definition 4.20 (Set of initial causes, multisets of added and dropped causes). Let $S = (E, \rightarrow, \triangleleft, \blacktriangleright)$ be a dynamic causality event structure with counters. Let $e \in E$ and $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$. Then:

1. The *set of initial causes of e* is defined by $\text{ic}(e) = \{e' \mid (e', e) \in \rightarrow\}$;
2. The *multiplicity of dropped causes of e after t* is defined by $\text{dc}(e, t) = \{e'^{(k)} \mid k = |\{e'' \in \bar{t} \mid e'' \triangleleft (e', e)\}|\}$;
3. The *multiplicity of added causes of e after t* is defined by $\text{ac}(e, t) = \{e'^{(k)} \mid k = |\{e'' \in \bar{t} \mid e'' \blacktriangleright (e', e)\}|\}$.

Definition 4.21 (DCCES proving sequence). Let $S = (E, \rightarrow, \triangleleft, \blacktriangleright)$ be a dynamic causality event structure with counters. A proving sequence of S is a trace $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$ satisfying the property: $\forall i \in [1, n] : \forall e \in E :$

$$\text{mult}(e, \text{ic}(e_i)) + \text{mult}(e, \text{ac}(e_i, t_{i-1})) - \text{mult}(e, \text{dc}(e_i, t_{i-1})) \geq 1 \implies e \in \overline{t_{i-1}}$$

Finally, we define proving sequences for RCESs directly, in agreement with the RCES semantics in [GP04].

Definition 4.22 (RCES proving sequence). Let $S = (E, \vdash)$ be an event structure for resolvable conflict. A proving sequence of S is a trace $t = e_1 \dots e_n$ with $\bar{t} \subseteq E$ satisfying the property:

$$\forall i \in [1, n] : \forall Z \subseteq \bar{t}_i : \exists W \subseteq \overline{t_{i-1}} : W \vdash Z$$

Informally, the securing property for RCESs means that every subset of events Z at timestamp i must be enabled by a subset of events W at timestamp $i - 1$.

Having introduced the notion of proving sequence for each class of ESs, we may now uniformly define the notion of configuration and a transition system on configurations for all kinds of ESs.

Definition 4.23 (Configuration and transition system). Let S be any ES. A *configuration* of S is any set X such that $X = \bar{t}$ for some proving sequence t of S . The *set of configurations* of S is $C(S) = \{\bar{t} \mid t \text{ is a proving sequence of } S\}$. The *multi-step transition relation* $\Rightarrow_S \subseteq C(S) \times C(S)$ on configurations is then defined as: $X \Rightarrow_S Y$ if there exist proving sequences t_1 and t_2 of S such that $\bar{t}_1 = X$ and $\bar{t}_2 = Y$ and t_1 is a prefix of t_2 . We then derive the *single action transition relation* \mapsto_S as: $X \mapsto_S Y$ if $X \Rightarrow_S Y$ and $|Y| = |X| + 1$.

We state some simple properties of proving sequences, to use in the proof of [Theorem 4.26](#), and define transition equivalence generically on any type of ES.

Lemma 4.24. *Let S be an ES of any of the aforementioned classes. Then:*

1. *If t is a proving sequence of S , then any prefix t_i of t is also a proving sequence of S ;*
2. *Let t and t' be two proving sequences of S such that $\bar{t} = \bar{t}'$. If te is a proving sequence of S , then also $t'e$ is a proving sequence of S .¹⁹*

¹⁹We note that this is true for any class of ESs in which the current configuration (i.e.,

Definition 4.25 (ES transition equivalence [GP04]). Two ESs S_1 and S_2 are called *multi-step transition equivalent*, written $S_1 \simeq_{mt} S_2$, if $\Rightarrow_{S_1} = \Rightarrow_{S_2}$. Analogously, two ESs S_1 and S_2 are called *single action transition equivalent*, written $S_1 \simeq_t S_2$, if $\mapsto_{S_1} = \mapsto_{S_2}$.

We conclude by showing that, using these semantics, any type of ES considered in this chapter can be encoded as an RCES. In particular, DCCESs also represent a subset of RCESs, while DCESs and RCESs have been shown to be incomparable [Arb+18].

Theorem 4.26. *Let S be an ES of any of the aforementioned classes, with set of events E , and let \mapsto_S be the corresponding transition relation as given by Definition 4.23. Then there exists an RCES $\widehat{S} = (E, \vdash)$ such that $\mapsto_{\widehat{S}} = \mapsto_S$.*

Proof. First we use the proving sequences of S to define \vdash . For any proving sequence $t = e_1 \dots e_n$ of S , define

$$\vdash_t = \{(\overline{t_{i-1}}, Z_i) \mid Z_i \subseteq \overline{t_i}, i \in 1, \dots, n\}.$$

Notice that for any t , we have $(\emptyset, \overline{t_1}) \in \vdash_t$. Now let

$$\vdash = \bigcup \{\vdash_t \mid t \text{ is a proving sequence of } S\}.$$

We proceed to show that S and \widehat{S} have the same sets of proving sequences.

- Let $t = e_1 \dots e_n$ be a proving sequence of S . Then, for any $1 \leq i \leq n$ and for any $Z_i \subseteq \overline{t_i}$, we have $\overline{t_{i-1}} \vdash_t Z_i$. It follows from Definition 4.22 that t is then also a proving sequence of \widehat{S} .
- We prove now that any proving sequence $t = e_1 \dots e_n$ of \widehat{S} is also a proving sequence of S . We proceed by induction on the length n of t .

For $n = 0$ we have $t = t_0 = \varepsilon$, which is a proving sequence of S by definition. Assume now that the statement holds for all proving sequences of length k , for $0 \leq k \leq n - 1$. We want to show that it

the set of occurred events) wholly determines the current causal state (i.e., which events are enabled). In particular, this is true for DCCESs but not for DCESs, which is why Theorem 4.26 does not hold for the latter.

also holds for $t = e_1 \dots e_n = t_{n-1}e_n$. By induction t_{n-1} is a proving sequence of S .

Since t is a proving sequence of \widehat{S} , by [Definition 4.22](#) for all $Z \subseteq \bar{t}$ there exists some $W \subseteq \overline{t_{n-1}}$ such that $W \vdash Z$. In particular this holds for $Z = \bar{t}$. Note that, on the one hand, $W \subseteq \overline{t_{n-1}}$ implies that $|W| \leq n - 1$. On the other hand, the construction of \vdash implies that, if $W \vdash Z$, then $|Z| < |W|$. Since $|\bar{t}| = n$, it follows that if $W \vdash \bar{t}$ then it must be $|W| = n - 1$ and thus $W = \overline{t_{n-1}}$. In other words: $\overline{t_{n-1}} \vdash_{t'} \bar{t}$ for some proving sequence t' of S .

Then, by definition of $\vdash_{t'}$, $\overline{t_{n-1}} = \overline{t'_{n-1}}$ and $\bar{t} \subseteq \overline{t'_n}$. Since $t = t_n$ and t_n and t'_n have the same length, it follows that $\bar{t} = \overline{t'_n}$. This means that $t'_n = t'_{n-1}e_n$. Now, since t'_n and t'_{n-1} are prefixes of t' , by [Lemma 4.24\(1\)](#) they are also proving sequences of S . Then we may use [Lemma 4.24\(2\)](#) to conclude that $t = t_n = t_{n-1}e_n$ is a proving sequence of S .

Since S and \widehat{S} have the same proving sequences, it directly follows from [Definition 4.23](#) that $\mapsto_S = \mapsto_{\widehat{S}}$. \square

4.3.2 Comparison – Overview

We now compare branching pomsets with the various classes of ESs previously reviewed. We establish a number of relative expressiveness results, summarised in [Figure 4.5](#), where we complete the initial picture of [Figure 4.4](#) with dashed red lines representing the non-inclusion of one model into another. Non-inclusion results are proved by providing counterexamples. The inclusion of tree-like BPs into PESs is proved in [Lemma 4.32](#). The inclusion of general BPs into RCESs is proved in [Theorem 4.37](#). The inclusion of general BPs into DCCESs is work in progress and for now remains a conjecture.

To conduct this comparison, we first need to introduce configurations also on branching pomsets, as well as a transition relation between them.

Definition 4.27 (Configuration). Let $R = (E, \preceq, \lambda, \mathcal{B})$ be a branching pomset. Then $X \subseteq E$ is a configuration of R if there exists some trace $t = e_1 \dots e_n$ such that $R \xrightarrow{t}^* R'$ and $\bar{t} = X$.

Just as for ESs, let $C(R)$ be the set of configurations of R .

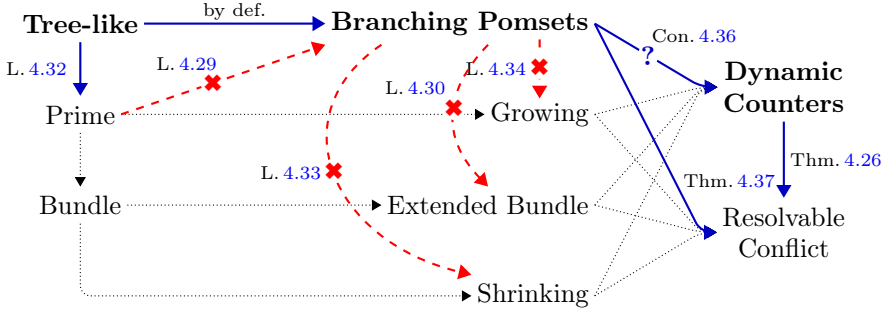


Figure 4.5: Summary of the core results proven in this section; solid blue arrows \longrightarrow represent strict inclusion, and dashed lines $- \times \blacktriangleright$ represent non-inclusion. A question mark indicates a conjecture. Labels are active pointers to Lemmas, Theorems and Conjectures.

Definition 4.28 (Transition system). Let $R = (E, \preceq, \lambda, \mathcal{B})$ be a branching pomset and let $X \subseteq E$. Then the transition relation $\mapsto_R \subseteq C(R) \times C(R)$ is defined as follows: $X \mapsto_R X \cup \{e\}$ if $R \xrightarrow{t}^* R' \xrightarrow{e}$ for some $t = e_1 \dots e_n$ such that $\bar{t} = X$.

We may now proceed to prove the results and discuss the conjectures corresponding to the solid blue arrows and dashed red arrows in Figure 4.5.

4.3.3 Comparison – Static models

Our first result states that the class of prime ESs is not included in that of BPs. Since prime ESs are the simplest class of ESs (static and dynamic alike), extended by all the others, this implies that no class of ESs is included in that of BPs. The PES in Figure 4.6a, which is used as a counterexample, is similar to the pomset “N” as described by Gischer [Gis88]. “N”, named after its shape, characterises the pomsets that are not expressible by sequential and parallel composition. It might be interesting to investigate whether Lemma 4.29 could be extended to an analogous characterisation of conflict relations, with choices instead of parallel composition.

Lemma 4.29 (PES $\not\subseteq$ BP). *There exists a prime event structure S for which there does not exist a branching pomset R such that $C(S) = C(R)$.*

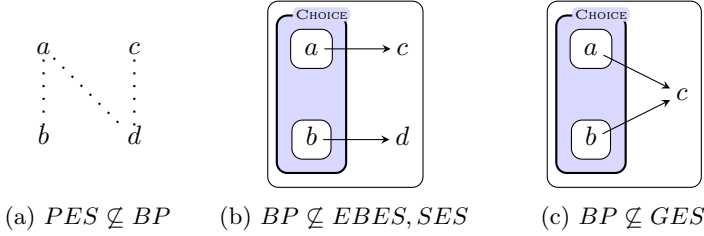


Figure 4.6: Counterexamples

Proof. Let $S = (E, \#, \leq)$ be the PES in Figure 4.6a, where $E = \{a, b, c, d\}$, $a \# b, a \# d, c \# d$ and $a \leq c, b \leq b, c \leq c, d \leq d$. Assume, for the sake of contradiction, that there exists some BP $R = (E, \preceq, \lambda, \mathcal{B})$ with the same set of configurations, namely $C(R) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, c\}, \{b, c\}, \{b, d\}\}$. Then, $a \# b$ implies that there exists some choice $\mathcal{C}_1 = \{\mathcal{B}_1, \mathcal{B}_2\}$ such that $a \succ \mathcal{B}_1$ and $b \succ \mathcal{B}_2$. Analogously, $c \# d$ implies that there exists $\mathcal{C}_2 = \{\mathcal{B}_3, \mathcal{B}_4\}$ such that $c \succ \mathcal{B}_3$ and $d \succ \mathcal{B}_4$.

Now there are four possible ways to relate \mathcal{C}_1 and \mathcal{C}_2 in \mathcal{B} . We show that all of them lead to a contradiction.

1. Suppose \mathcal{C}_1 and \mathcal{C}_2 are concurrent choices: $\mathcal{B} = \{\mathcal{C}_1, \mathcal{C}_2\}$. Then a and d can occur together, which is a contradiction.
2. Suppose \mathcal{C}_1 and \mathcal{C}_2 are nested choices. Let $\mathcal{C}_2 \succ \mathcal{C}_1$ (the symmetric case is analogous). There are two subcases:
 - (a) either $\mathcal{C}_2 \succ \mathcal{B}_1$, in which case b (which is in \mathcal{B}_2) can never occur together with c or d (which are then in \mathcal{B}_1), thus contradicting $\{b, c\} \in C(R)$;
 - (b) or $\mathcal{C}_2 \succ \mathcal{B}_2$, in which case a (which is in \mathcal{B}_1) can never occur together with c or d (which are then in \mathcal{B}_2), thus contradicting $\{a, c\} \in C(R)$;
3. Suppose \mathcal{C}_1 and \mathcal{C}_2 are the same choice, namely $\{\mathcal{B}_1, \mathcal{B}_2\} = \{\mathcal{B}_3, \mathcal{B}_4\}$. Then, either $\mathcal{B}_1 = \mathcal{B}_3$, in which case b (which is in $\mathcal{B}_2 = \mathcal{B}_4$) cannot occur together with c (which is in $\mathcal{B}_1 = \mathcal{B}_3$), thus contradicting $\{b, c\} \in C(R)$; or $\mathcal{B}_1 = \mathcal{B}_4$, in which case a (which is in $\mathcal{B}_1 = \mathcal{B}_4$) cannot occur together with c (which is in $\mathcal{B}_2 = \mathcal{B}_3$), thus contradicting

$$\{a, c\} \in C(R);$$

4. Suppose \mathcal{C}_1 and \mathcal{C}_2 are mutually exclusive choices. Then, there must be some choice $\mathcal{C} = \{\mathcal{B}_5, \mathcal{B}_6\}$ such that $\mathcal{C}_1 \succ \mathcal{B}_5$ and $\mathcal{C}_2 \succ \mathcal{B}_6$. Consequently, a (which is in $\mathcal{B}_1 \succ \mathcal{B}_5$) cannot occur together with c (which is in $\mathcal{B}_3 \succ \mathcal{B}_6$), thus again contradicting $\{a, c\} \in C(R)$.

Since all cases are contradictory, we conclude that there does not exist any BP R such that $C(R) = C(S)$. \square

The following lemma states that the class of BPs is not included in the class of EBESs. Intuitively, it is possible for a BP in which a certain event can be enabled, to reduce to one in which it cannot, and to then reduce to one in which it again can. This cannot be simulated by an EBES, since disabling through asymmetric conflict cannot be undone. Since EBESs are the most expressive class of static ESs in [Figure 4.5](#) (it includes the class of PESs and BESs), this implies that no class of static ESs in [Figure 4.5](#) includes that of BPs.

Lemma 4.30 (*BP $\not\subseteq$ EBES*). *There exists a branching pomset R for which there does not exist an extended bundle event structure S such that $C(R) = C(S)$.*

Proof. Let R be the BP in [Figure 4.6b](#). Assume, for the sake of contradiction, that there exists some EBES $S = (E, \rightsquigarrow, \succ)$ with the same set of configurations, namely $C(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{a, c, d\}, \{b, c, d\}\}$. Since $\{a\}, \{b\}, \{c\}, \{d\} \in C(S)$, it follows that $\succ = \emptyset$. Furthermore, since $\{a, c, d\} \in C(S)$, it follows that $\neg(c \rightsquigarrow d) \vee \neg(d \rightsquigarrow c)$. However, it follows from these that $\{c, d\} \in C(S)$, which is a contradiction. \square

The following theorem follows directly from [Lemmas 4.29](#) and [4.30](#).

Theorem 4.31. *Branching pomsets are incomparable with the static event structures in [Figure 4.5](#) (prime, bundle, and extended bundle).*

In contrast, for the special class of tree-like BPs, we can prove that it is strictly subsumed by the class of PESs. Tree-like BPs are formally introduced as part of the well-formedness conditions in [Chapter 6](#). Intuitively, a BP is tree-like if events inside of choices can only affect future events in the same branch; graphically speaking, arrows can only enter boxes and not leave them.

Lemma 4.32. *For every tree-like branching pomset R there exists a prime event structure S such that $C(R) = C(S)$.*

Proof. Let $R = (E, \preceq, \lambda, \mathcal{B})$ be a BP. We construct the PES $S = (E, \#, \leq)$, where $\leq = \preceq^*$, the reflexive and transitive closure of \preceq , and $\# = \{(e_1, e_2) \mid \exists \{\mathcal{B}_1, \mathcal{B}_2\} \succ \mathcal{B} : e_1 \succ \mathcal{B}_1 \wedge e_2 \succ \mathcal{B}_2\}$, i.e., the set of conflicts consists exactly of all pairs of events which are separated by some choice. Since R is tree-like, $e \succ \mathcal{B}' \wedge e \preceq e' \implies e' \succ \mathcal{B}'$, from which it follows that S satisfies conflict hereditariness. We proceed by showing that R and S have the same traces and thus the same configurations.

- Let $t = e_1 \dots e_n$ be a trace of R . Then $R \xrightarrow{t}^* R'$ for some R' . Since two mutually exclusive events can never occur in the same trace of R , t satisfies the PES consistency condition. Furthermore, for all $i \in \{1, \dots, n\}$, if there exists some $e \prec e_i$ then, since R is tree-like, $e \in \overline{t_{i-1}}$. In other words, t also satisfies the PES left-closure condition, and then t is also a trace of S .
- Let $t = e_1 \dots e_n$ be a trace of S . Then t must be consistent and left-closed. Let $i \in \{1, \dots, n\}$ and assume that $R \xrightarrow{t_{i-1}}^* R'$ for some R' (where $t_0 = \varepsilon$, in which case $R' = R$). Since t is left-closed, it follows that, for every e such that $e \leq e_i$, we have $e \in \overline{t_{i-1}}$. Consequently, e_i is minimal in R' . Furthermore, since t is consistent, e_i cannot be in conflict with any event in t_{i-1} and thus also does not belong to a different branch of any choice than the events in t_{i-1} . Since R is tree-like, there is no need to resolve choices for any reason other than firing events in them. It follows that $e_i \succ R'$, and then $R' \xrightarrow{e_i} R''$ for some R'' . Finally, it then follows by induction that $R \xrightarrow{t}^* \hat{R}$ for some \hat{R} and then t is a trace of R .

Since R and S have the same set of traces, it follows that $C(R) = C(S)$. \square

4.3.4 Comparison – Dynamic models

Our next two lemmas state that the class of BPs is not included in the classes of SESs and GESs. Intuitively, BPs are able to disable events, which cannot be done by SESs, and they can model disjunctive causality, which cannot be done by GESs. Combined with our result in the previous subsection that the class of PESs (subsumed by those of SESs and GESs) is not included in the class of BPs, we conclude that BPs and SESs/GESs are incomparable: the expressive

power to only remove, or to only add, causalities dynamically is insufficient to cover the expressive power of BPs, and vice versa.

Lemma 4.33 (*BP $\not\subseteq$ SES*). *There exists a branching pomset R for which there does not exist a shrinking causality event structure S such that $C(R) = C(S)$.*

Proof. Let R be the BP in [Figure 4.6b](#). Assume, for the sake of contradiction, that there exists some SES $S = (E, \#, \rightarrow, \triangleleft)$ with the same set of configurations, namely $C(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{a, c, d\}, \{b, c, d\}\}$. Since $\{a\}, \{b\}, \{c\}, \{d\} \in C(S)$, it follows that $\rightarrow = \emptyset$. Furthermore, since $\{a, c, d\} \in C(S)$, it follows that $\neg(c \# d)$. However, it follows from these that $\{c, d\} \in C(S)$, which is a contradiction. \square

Lemma 4.34 (*BP $\not\subseteq$ GES*). *There exists a branching pomset R for which there does not exist a growing causality event structure S such that $C(R) = C(S)$.*

Proof. Let R be the BP in [Figure 4.6c](#). Assume, for the sake of contradiction, that there exists some GES $S = (E, \rightarrow, \blacktriangleright)$ with the same set of configurations, namely $C(S) = \{\emptyset, \{a\}, \{b\}, \{a, c\}, \{b, c\}\}$. Since $\{c\} \notin C(S)$ and $\{a, c\} \in C(S)$, it follows that $a \rightarrow c$. However, since GESs cannot model disjunctive causality, it follows from $a \rightarrow c$ that $\{b, c\} \notin C(S)$, which is a contradiction. \square

The following theorem follows directly from [Lemmas 4.29](#), [4.33](#) and [4.34](#).

Theorem 4.35. *Branching pomsets are incomparable with two dynamic event structures in [Figure 4.5](#) (growing and shrinking).*

In contrast, we conjecture that DCCESs (which combine the power of SESs and GESs) have more expressive power than BPs. The idea is that the power to remove dependencies can be used to encode disjunctive causality, while the power to add dependencies can be used to encode partial termination (as asymmetric conflicts).

Conjecture 4.36. For every branching pomset R there exists a dynamic causality event structure with counters S such that $R \simeq_t S$.

Surprisingly, we can show that BPs are incomparable with the original semantics of DCEs. In [Appendix B](#) we give an example of a BP for which there exists no equivalent DCE. We note that this BP can be encoded as a DCE and that this counterexample thus does not invalidate [Conjecture 4.36](#).

Finally, the general enabling relation of RCEs can essentially encode arbitrary transition graphs, including those induced by BPs. When equating traces of BPs to proving sequences for ESs, their definitions for configurations and (single action) transition relations ([Definitions 4.23](#), [4.27](#) and [4.28](#)) coincide. Then, occasionally substituting the word ‘trace’ for ‘proving sequence’ in the proof of [Theorem 4.26](#) also proves the following theorem.

Theorem 4.37. For every branching pomset R there exists an event structure for resolvable conflict S such that $R \simeq_t S$.

4.4 Conclusion

This chapter introduced branching pomsets as a compact model for combinations of concurrency and choice, and compared it with the existing literature on event structures. The branching structure of BPs enforces a strict hierarchy on choices, which is less expressive as a choice mechanism than the conflict relation typical to ESs. However, the refinement and enabling relations of BPs allow for (perhaps surprisingly) complex behaviour, placing it in the realm of dynamic ESs. Consequently, none of the existing classes of ESs are contained in BPs, but neither are BPs contained in many classes of ESs: they are incomparable with most, and are only contained in some of the most expressive classes of dynamic ESs. Specifically, BPs are not contained in dynamic causality ESs and, consequently, also not in extended bundle ESs, or in growing or shrinking causality ESS; they are, however, contained in ESs for resolvable conflict, and we conjecture that they are contained in a variant of dynamic causality ESs. This partially answers [Question 2](#); [Chapter 5](#) completes the answer by specifically applying BPs to choreographies.

We end with a brief discussion on properties of BPs and related models.

n -ary choices The definition of our branching structure ([Definition 4.1](#)) only supports binary choices: n -ary choices only exist as a derived concept by

nesting binary ones. This matches the structure of typical choreographies, but it would be more natural to represent, e.g., $c_1 + (c_2 + c_3)$ (where $+$ represents nondeterministic choice) as a single choice between the BPs for c_1 , c_2 and c_3 instead of as two nested binary choices, e.g., a choice between, on the one hand, c_1 and, on the other hand, a choice between c_2 and c_3 . Supporting n -ary choices could be realised in two ways. On the one hand, n -ary choices could be supported in encodings but then internally (pre)processed as nested binary choices, as they are now; this can be considered syntactic sugar. On the other hand, n -ary choices could be added as a primitive by altering [Definition 4.1](#), of which binary choices would then be a special case. The latter would not change the expressiveness of BPs, but is intellectually more interesting: it would also require some thought about how to change the rules for refinement ([Figure 4.2a](#)), in particular CHOICE. A naive change would be to simply have this rule use $i \in \{1, \dots, n\}$ and $\{\{\mathcal{B}_1, \dots, \mathcal{B}_n\}\}$ instead of its current binary rules, but this is not sufficient as this naive n -ary choice would not be equivalent to the same branches composed as nested binary choices. For example, the previously described nested choice can refine to a choice between c_1 and c_2 (by resolving the inner choice between c_2 and c_3 in favour of c_2), but a BP whose branching structure consists of a single ternary choice $\{\{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3\}\}$ would not be able to refine to $\{\{\mathcal{B}_1, \mathcal{B}_2\}\}$ as the rules would only allow it to either refine all of its branches or discard all but one of them. Supporting a primitive n -ary choice would thus also require a new rule that allows $\{\{\mathcal{B}_1, \dots, \mathcal{B}_n\}\}$ to refine a choice to an arbitrary (non-empty) subset of its branches, and one to refine singleton choices to their sole branch.

Partial order In [Definition 4.4](#), \preceq is defined as a relation on events such that its transitive closure is a partial order, rather than \preceq being a partial order itself as \leq is in traditional pomsets. This is necessary because two events may be either related or unrelated depending on the resolution of certain choices. For example, consider the BP R_a in [Figure 4.3](#). Events a and d are related if the choice's upper branch is taken and unrelated otherwise. It should thus be possible to enable d by discarding the choice's upper branch. In our current rules d only has a direct dependency on b and therefore discarding b will make d minimal. However, if \preceq were a partial order, then d would also have direct dependencies on a and, since a is not part of the choice, there would be no refinement that enables d .

In general, if $R_1 \sqsupseteq R'_1$ and $R_2 \sqsupseteq R'_2$ then if \preceq would be a partial order it would not necessarily be true that $R_1 ; R_2 \sqsupseteq R'_1 ; R'_2$, where the sequential composition ‘;’ of two BPs is as later defined in [Figure 5.4](#). $R_1 ; R_2$ may contain dependencies

obtained by transitivity which would still be present in the refinement but which cannot be derived in $R'_1 ; R'_2$. Castellani and Zhang note the same property in the context of flow event structures [CZ97].

Relation between BPs and ESs Figure 4.5 shows the established inclusions and non-inclusions between BPs and ESs. Naturally, formally proving or disproving the inclusion of BPs in our variant of dynamic causality ESs (Conjecture 4.36) would complete the picture. A comparison with the original semantics of dynamic causality ESs may be found in Appendix B. Additionally, it may be interesting to further study the difference in expressiveness between BPs and the classes of ESs. As demonstrated by Lemma 4.29, (prime) ESs can describe more complex choices than BPs. A way to close or overcome the gap would be to lift the requirement on a BP's branching structure that all events must occur in it exactly once, thus allowing overlapping boxes in the graphical representation. This would also require changes to the refinement rules. As the branching structure could then encode a (symmetric) conflict relation, this would invalidate Lemma 4.29. It is unclear precisely how it would affect the other relations.

Relation between BPs and other models We have given a thorough comparison between BPs and several classes of ESs, since they share many ideas. It would be interesting to also investigate the relation between BPs and other models for concurrency, with the book chapter by Winskel and Nielsen being a good starting point [WN93]. In particular, Petri nets are a well-established model for concurrency and bear a close relationship with ESs, and they are a first candidate for further study.

Branching pomsets for choreographies

This chapter is based on section 4 of our ICE 2022 paper [ICE22] and on section 4 of the subsequent JLAMP 2024 paper [JLAMP24]. It partially answers [Question 2](#). As in the original papers, a number of technical lemmas and proofs are omitted in favour of informal proof sketches or highlights. The omitted proofs and lemmas can be found in the technical report of our ICE 2022 paper [ICE22 TR].

5.1 Introduction

[Chapter 4](#) introduced branching pomsets (BPs) as a generic model for concurrency. In this chapter we define the syntax and semantics of a simple choreography language, for which we provide an encoding as BPs. We then prove that the semantics of the choreographies and their encodings as BPs coincide — specifically, their LTSs are bisimilar.

While the presented choreography language is conceptually simple, its weak sequential composition is an interesting feature. Our interpretation of it, and the implementation through partial termination, are adapted from the work of Rensink and Wehrheim in process algebra [RW01]. It is novel in the context of choreographies.

$$c ::= \mathbf{1} \mid \mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} \mid \boxed{\mathbf{ab} ? \mathbf{x}} \mid c ; c \mid c + c \mid c \parallel c \mid c^*$$

Figure 5.1: Syntax of choreographies, where \mathbf{a} and \mathbf{b} are processes ($\mathbf{a} \neq \mathbf{b}$) and \mathbf{x} is a message type.

Formally, we fix the following set of labels for BPs for this chapter. Let $\mathcal{A} = \{\mathbf{a}, \mathbf{b}, \dots\}$ be the set of processes and let $\mathcal{X} = \{\mathbf{x}, \mathbf{y}, \dots\}$ be the set of message types. Then, let $\mathcal{L} = \bigcup_{\mathbf{a}, \mathbf{b} \in \mathcal{A}, \mathbf{x} \in \mathcal{X}} \{\mathbf{ab} ! \mathbf{x}, \mathbf{ab} ? \mathbf{x}\}$ be the set of labels (actions), ranged over by ℓ , where $\mathbf{ab} ! \mathbf{x}$ is a send action from \mathbf{a} to \mathbf{b} of a message of type \mathbf{x} , and $\mathbf{ab} ? \mathbf{x}$ is the corresponding receive action by \mathbf{b} . The *subject* of an action ℓ , written $subj(\ell)$, is the process performing the action: $subj(\mathbf{ab} ! \mathbf{x}) = \mathbf{a}$ and $subj(\mathbf{ab} ? \mathbf{x}) = \mathbf{b}$.

5.2 Choreography language definition

The syntax of our choreography language is formally defined in Figure 5.1. ‘ $\mathbf{1}$ ’ is the empty choreography; ‘ $\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x}$ ’ is the asynchronous communication from \mathbf{a} to \mathbf{b} of a message of type \mathbf{x} ; the boxed term ‘ $\mathbf{ab} ? \mathbf{x}$ ’ represents a pending receive on \mathbf{b} from \mathbf{a} of a message of type \mathbf{x} (it is boxed in Figure 5.1 to indicate that it is only used internally to formalise behaviour but the box is not part of the syntax); ‘ $c_1 ; c_2$ ’, ‘ $c_1 + c_2$ ’ and ‘ $c_1 \parallel c_2$ ’ are respectively the weak sequential composition, nondeterministic choice and parallel composition of choreographies c_1 and c_2 ; finally, ‘ c^* ’ is the finite repetition (or, more informally, loop) of choreography c .

The semantics for choice, parallel composition and loop are standard. We note that our sequential composition is weak. With standard sequential composition, when sequencing c_1 and c_2 , the choreography c_1 must fully terminate before proceeding to c_2 . With weak sequential composition, however, actions in c_2 can already be executed as long as they do not interfere with c_1 . For example, in $\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} ; \mathbf{c} \rightarrow \mathbf{d} : \mathbf{x}$ we can execute the action $\mathbf{cd} ! \mathbf{x}$ as it does not affect the subjects of $\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x}$: there is no dependency and thus no need to wait for $\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x}$ to go first. On the other hand, in $\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} ; \mathbf{a} \rightarrow \mathbf{c} : \mathbf{x}$ the action $\mathbf{ac} ! \mathbf{x}$ cannot be executed first as its subject (\mathbf{a}) must first execute $\mathbf{ab} ! \mathbf{x}$. This is the common interpretation of sequential composition in the context of message sequence charts [KL98], multiparty session types [HYC08] and choreography programming [CM13].

The reduction and termination rules of our choreography language are formally defined in Figures 5.2a and 5.2b, respectively. To formalise the reduction of weak sequential composition, we follow Rensink and Wehrheim [RW01], who define a

$$\begin{array}{c}
\frac{}{\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} \xrightarrow{\mathbf{ab}! \mathbf{x}} \mathbf{ab} ? \mathbf{x}} [\rightarrow_1] \quad \frac{}{\mathbf{ab} ? \mathbf{x} \xrightarrow{\mathbf{ab} ? \mathbf{x}} \mathbf{1}} [\rightarrow_2] \quad \frac{c_1 \xrightarrow{\ell} c'_1}{c_1 ; c_2 \xrightarrow{\ell} c'_1 ; c_2} [\rightarrow_3] \\
\frac{c_1 \xrightarrow{\check{\ell}} c'_1 \quad c_2 \xrightarrow{\ell} c'_2}{c_1 ; c_2 \xrightarrow{\ell} c'_1 ; c'_2} [\rightarrow_4] \quad \frac{c_1 \xrightarrow{\ell} c'_1}{c_1 \parallel c_2 \xrightarrow{\ell} c'_1 \parallel c_2} [\rightarrow_5] \quad \frac{c_2 \xrightarrow{\ell} c'_2}{c_1 \parallel c_2 \xrightarrow{\ell} c_1 \parallel c'_2} [\rightarrow_6] \\
\frac{c_1 \xrightarrow{\ell} c'_1}{c_1 + c_2 \xrightarrow{\ell} c'_1} [\rightarrow_7] \quad \frac{c_2 \xrightarrow{\ell} c'_2}{c_1 + c_2 \xrightarrow{\ell} c'_2} [\rightarrow_8] \quad \frac{c \xrightarrow{\ell} c'}{c^* \xrightarrow{\ell} c' ; c^*} [\rightarrow_9]
\end{array}$$

(a) Reduction rules.

$$\frac{}{\mathbf{1} \downarrow} [\downarrow_1] \quad \frac{}{c^* \downarrow} [\downarrow_2] \quad \frac{c_1 \downarrow \quad c_2 \downarrow \quad \dagger \in \{;, \parallel\}}{c_1 \dagger c_2 \downarrow} [\downarrow_3] \quad \frac{c_i \downarrow \quad i \in \{1, 2\}}{c_1 + c_2 \downarrow} [\downarrow_4]$$

(b) Termination rules.

$$\begin{array}{c}
\frac{}{\mathbf{1} \xrightarrow{\check{\ell}} \mathbf{1}} [\check{\rightarrow}_1] \quad \frac{\text{subj}(\ell) \notin \{\mathbf{a}, \mathbf{b}\}}{\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} \xrightarrow{\check{\ell}} \mathbf{a} \rightarrow \mathbf{b} : \mathbf{x}} [\check{\rightarrow}_2] \quad \frac{\text{subj}(\ell) \neq \mathbf{b}}{\mathbf{ab} ? \mathbf{x} \xrightarrow{\check{\ell}} \mathbf{ab} ? \mathbf{x}} [\check{\rightarrow}_3] \\
\frac{c_1 \xrightarrow{\check{\ell}} c'_1 \quad c_2 \xrightarrow{\check{\ell}} c'_2 \quad \dagger \in \{;, \parallel, +\}}{c_1 \dagger c_2 \xrightarrow{\check{\ell}} c'_1 \dagger c'_2} [\check{\rightarrow}_4] \\
\frac{c_1 \xrightarrow{\check{\ell}} c'_1 \quad c_2 \not\xrightarrow{\check{\ell}}}{c_1 + c_2 \xrightarrow{\check{\ell}} c'_1} [\check{\rightarrow}_5] \quad \frac{c_1 \not\xrightarrow{\check{\ell}} \quad c_2 \xrightarrow{\check{\ell}} c'_2}{c_1 + c_2 \xrightarrow{\check{\ell}} c'_2} [\check{\rightarrow}_6] \quad \frac{c \xrightarrow{\check{\ell}} c}{c^* \xrightarrow{\check{\ell}} c^*} [\check{\rightarrow}_7] \quad \frac{c \not\xrightarrow{\check{\ell}} c}{c^* \xrightarrow{\check{\ell}} \mathbf{1}} [\check{\rightarrow}_8]
\end{array}$$

(c) Partial termination rules.

Figure 5.2: Operational semantics of choreographies.

notion of *partial termination*. Partial termination inspired our refinement and enabling rules for BPs, so both the concepts and notation are similar.

Partial termination In a weak sequential composition $c_1 ; c_2$, an action ℓ in c_2 can be executed if c_1 can *partially terminate* for ℓ . Conceptually, a choreography c_1 can partially terminate for ℓ by discarding all branches of its behaviour which would conflict with it, i.e., in which the subject of ℓ occurs.

This is written $c_1 \xrightarrow{\ell} c'_1$, where c'_1 is the remainder of c_1 after discarding all branches involving the subject of ℓ . For example, if $c_1 = \mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} + \mathbf{a} \rightarrow \mathbf{c} : \mathbf{x}$ then $c_1 \xrightarrow{\mathbf{c}d!x} \mathbf{a} \rightarrow \mathbf{b} : \mathbf{x}$, as this branch does not contain \mathbf{c} . An exception is when the subject of ℓ occurs in *every* branch of c_1 , in which case c_1 cannot partially terminate for ℓ , i.e., $c_1 \not\xrightarrow{\ell}$. In the above example, $c_1 \not\xrightarrow{\mathbf{a}d!x}$.

The rules for partial termination are deterministic and, like our enabling relation for BPs, only discard the absolutely necessary. In the example above, $c_1 \xrightarrow{\mathbf{d}a!x} c_1$ since the subject \mathbf{d} does not occur in either branch: dropping one of the branches would be unnecessary and is thus not allowed. The rules for partial termination are defined in Figure 5.2c. We highlight the rules for the different operators:

- Sequential composition $c_1; c_2$ and parallel composition $c_1 \parallel c_2$ can partially terminate if both c_1 and c_2 can (rule $\xrightarrow{4}$).
- A choice $c_1 + c_2$ can partially terminate if at least one of its branches can. If both branches can partially terminate then both are kept (rule $\xrightarrow{4}$), otherwise only the partially terminated one is kept (rules $\xrightarrow{5}$ and $\xrightarrow{6}$).
- Following Rensink and Wehrheim, a loop c^* can partially terminate if its body (c) can partially terminate without discarding any branches, i.e., if $c \xrightarrow{\ell} c$. In that case also $c^* \xrightarrow{\ell} c^*$ (rule $\xrightarrow{7}$). Otherwise we allow c^* to be skipped entirely, represented as partial termination to $\mathbf{1}$, i.e., $c^* \xrightarrow{\ell} \mathbf{1}$ (rule $\xrightarrow{8}$). This can happen either if c can partially terminate to c' but $c' \neq c$, or if c cannot partially terminate at all. We use $c \not\xrightarrow{\ell} c$ as a shorthand to cover both these cases. Skipping a loop is necessary, for example, in a protocol such as $(\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} ; \mathbf{b} \rightarrow \mathbf{a} : \mathbf{x})^* ; \mathbf{a} \rightarrow \mathbf{b} : \mathbf{done}$, in which Alice and Bob exchange an arbitrary number of messages \mathbf{x} until Alice signals **done**. In this choreography, the loop has to partially terminate to $\mathbf{1}$ to eventually allow for the action $\mathbf{ab}!done$.

Example 5.1. Let $c_1 \boxplus = (\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} + \mathbf{a} \rightarrow \mathbf{c} : \mathbf{x}) ; (\mathbf{d} \rightarrow \mathbf{b} : \mathbf{x} + \mathbf{d} \rightarrow \mathbf{e} : \mathbf{x})$. Let $c_2 \boxplus = (\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} + \mathbf{c} \rightarrow \mathbf{b} : \mathbf{x})^* \parallel (\mathbf{c} \rightarrow \mathbf{a} : \mathbf{x} + \mathbf{c} \rightarrow \mathbf{b} : \mathbf{x})$.

- $c_1 \xrightarrow{\mathbf{b}e!x} \mathbf{a} \rightarrow \mathbf{c} : \mathbf{x} ; \mathbf{d} \rightarrow \mathbf{e} : \mathbf{x}$. The subject \mathbf{b} of $\mathbf{b}e!x$ occurs in one branch of each of both choices: $\mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} + \mathbf{a} \rightarrow \mathbf{c} : \mathbf{x} \xrightarrow{\mathbf{b}e!x} \mathbf{a} \rightarrow \mathbf{c} : \mathbf{x}$ (rule $\xrightarrow{6}$) and $\mathbf{d} \rightarrow \mathbf{b} : \mathbf{x} + \mathbf{d} \rightarrow \mathbf{e} : \mathbf{x} \xrightarrow{\mathbf{b}e!x} \mathbf{d} \rightarrow \mathbf{e} : \mathbf{x}$ (rule $\xrightarrow{6}$), and then $c_1 \xrightarrow{\mathbf{b}e!x} \mathbf{a} \rightarrow \mathbf{c} : \mathbf{x} ; \mathbf{d} \rightarrow \mathbf{e} : \mathbf{x}$ (rule $\xrightarrow{4}$). While the recipient \mathbf{e} also occurs in the second branch of the second choice, since it is not the actual subject it does not interfere with $\mathbf{b}e!x$ (rule $\xrightarrow{2}$).

- $c_1 \not\check{\rightarrow}_{ab!x}$. While the second choice can partially terminate without dropping any branches (rule $\check{\rightarrow}_4$), the first choice contains the subject a of $ab!x$ in both of its branches and none of rules $\check{\rightarrow}_4$, $\check{\rightarrow}_5$ and $\check{\rightarrow}_6$ apply. Since one of the choices cannot partially terminate, neither can their sequential composition: rule $\check{\rightarrow}_4$ does not apply.
- $c_2 \check{\rightarrow}_{ad!x} \mathbf{1} \parallel c \rightarrow b:x$. The subject a of $ad!x$ only occurs in one branch of the loop body, but, since rule $\check{\rightarrow}_7$ does not apply, the loop can only partially terminate to $\mathbf{1}$ through rule $\check{\rightarrow}_8$. On the right hand side of the parallel composition, a occurs only in the first branch and so rule $\check{\rightarrow}_6$ applies. The two sides are then combined through rule $\check{\rightarrow}_4$.
- $c_2 \not\check{\rightarrow}_{cd!x}$. While the loop can again partially terminate to $\mathbf{1}$ through rule $\check{\rightarrow}_8$, the subject c of $cd!x$ occurs in both branches of the right hand side of the parallel composition and none of rules $\check{\rightarrow}_4$, $\check{\rightarrow}_5$ and $\check{\rightarrow}_6$ apply. Since its right hand side cannot partially terminate, neither can it as a whole: rule $\check{\rightarrow}_4$ does not apply.

As already discovered by Rensink and Wehrheim [RW01], an unwanted consequence of these rules for partial termination is that unfolding iterations of loops no longer preserves behaviour. We would like c^* and $(c; c^*) + \mathbf{1}$ to behave the same, but this is not the case. For example, if $c = a \rightarrow b:x + c \rightarrow d:x$, then $c^* \check{\rightarrow}_{ab!x} \mathbf{1}$ (rule $\check{\rightarrow}_8$) but $(c; c^*) + \mathbf{1} \check{\rightarrow}_{ab!x} (c \rightarrow d:x; \mathbf{1}) + \mathbf{1}$ (rules $\check{\rightarrow}_6$, $\check{\rightarrow}_8$ and $\check{\rightarrow}_4$). Then $c^*; c \check{\rightarrow}_{ab!x} \mathbf{1}; ab?x$ by skipping the loop (rule \rightarrow_4); however, $((c; c^*) + \mathbf{1}); c$ has no way to match this as it can skip the loop but it can only partially terminate the already unfolded iteration c to $c \rightarrow d:x$ — it cannot discard it entirely. We borrow the solution that Rensink and Wehrheim offer, which is the concept of *dependent guardedness*.

Dependent guardedness A loop c^* is *dependently guarded* if, for all actions ℓ , the loop body c can only partially terminate for ℓ if its subject does not occur anywhere in c . In other words: any process that occurs in some branch of c must also occur in every other branch of c . It then follows that c can either partially terminate for ℓ without having to discard any branches, or it cannot partially terminate at all. Formally: if $c \check{\rightarrow}_\ell c'$ then $c' = c$. A choreography \hat{c} is then dependently guarded if all of its loops are.

As a consequence, we avoid the problem above: if $c^* \check{\rightarrow}_\ell \mathbf{1}$ then $c \not\check{\rightarrow}_\ell$ and $c; c^* \not\check{\rightarrow}_\ell$ since rule $\check{\rightarrow}_4$ does not apply and, consequently, $(c; c^*) + \mathbf{1}$ is forced to partially terminate to the second branch of the choice (rule $\check{\rightarrow}_6$), which is $\mathbf{1}$. More precisely, let c^* be some dependently guarded expression. If

$c \xrightarrow{\check{\ell}} c'$ for some ℓ, c' , then $c' = c$. It follows that $c^* \xrightarrow{\check{\ell}} c^*$ (rule $\xrightarrow{\check{\ell}}_7$) and $(c; c^*) + \mathbf{1} \xrightarrow{\check{\ell}} (c; c^*) + \mathbf{1}$ (rule $\xrightarrow{\check{\ell}}_4$). Similarly, if $c \not\xrightarrow{\check{\ell}}$ then $c^* \xrightarrow{\check{\ell}} \mathbf{1}$ (rule $\xrightarrow{\check{\ell}}_8$) and $(c; c^*) + \mathbf{1} \xrightarrow{\check{\ell}} \mathbf{1}$ (rule $\xrightarrow{\check{\ell}}_6$).

Example 5.2. Let $c_1 \boxplus = \mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} + \mathbf{a} \rightarrow \mathbf{c} : \mathbf{x}$. Let $c_2 \boxplus = \mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} + \mathbf{b} \rightarrow \mathbf{a} : \mathbf{x}$.

- c_1 itself is dependently guarded as it does not contain any loop.
- $c_1^* \boxplus$ is not dependently guarded as $c_1 \xrightarrow{\check{\text{cd!x}}} \mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} \neq c_1$ (rule $\xrightarrow{\check{\ell}}_5$).
- $c_2^* \boxplus$ is dependently guarded since both \mathbf{a} and \mathbf{b} occur in both branches of c_2 .
- $(c_2^*)^* \boxplus$ is not dependently guarded, since $c_2^* \xrightarrow{\check{\text{ab!x}}} \mathbf{1}$ (rule $\xrightarrow{\check{\ell}}_8$).²⁰

5.3 BP encoding

Figure 5.3 shows two choreographies and corresponding BPs similar to those in Figure 4.1. Formally, the rules for the construction of a BP for a choreography c , written $\llbracket c \rrbracket$, are defined in Figure 5.4. Most rules are as expected. We highlight the rules for operators:

- The rule for parallel composition ($\llbracket c_1 \parallel c_2 \rrbracket$) takes the pairwise union of all components.
- The rule for sequential composition ($\llbracket c_1 ; c_2 \rrbracket$) also adds dependencies to ensure that, for every \mathbf{a} , all events with subject \mathbf{a} in $\llbracket c_1 \rrbracket$ (denoted $E_1^{\mathbf{a}}$) must precede all events with subject \mathbf{a} in $\llbracket c_2 \rrbracket$ ($E_2^{\mathbf{a}}$). This matches the reduction rule for weak sequential composition of choreographies (Figure 5.2a), as events in $\llbracket c_2 \rrbracket$ are only required to wait for events in $\llbracket c_1 \rrbracket$ whose subject is the same.
- The rule for choice ($\llbracket c_1 + c_2 \rrbracket$) adds a single top-level choice in the branching structure to choose between the BPs for c_1 and c_2 .
- The rule for loops ($\llbracket c^* \rrbracket$) encodes a loop as a choice between terminating ($\mathbf{1}$) and unfolding one iteration of the loop ($c; c^*$). This results in an infinitely unfolding BP, which is thus of infinite size. We note that our theoretical results still hold even on infinite BPs, but that any analysis of an infinite BP will have to be symbolic. However, since the focus of

²⁰In fact, such a double loop is only dependently guarded if it does not contain any communications, i.e., if it only consists of some number of $\mathbf{1}$ s and operators.

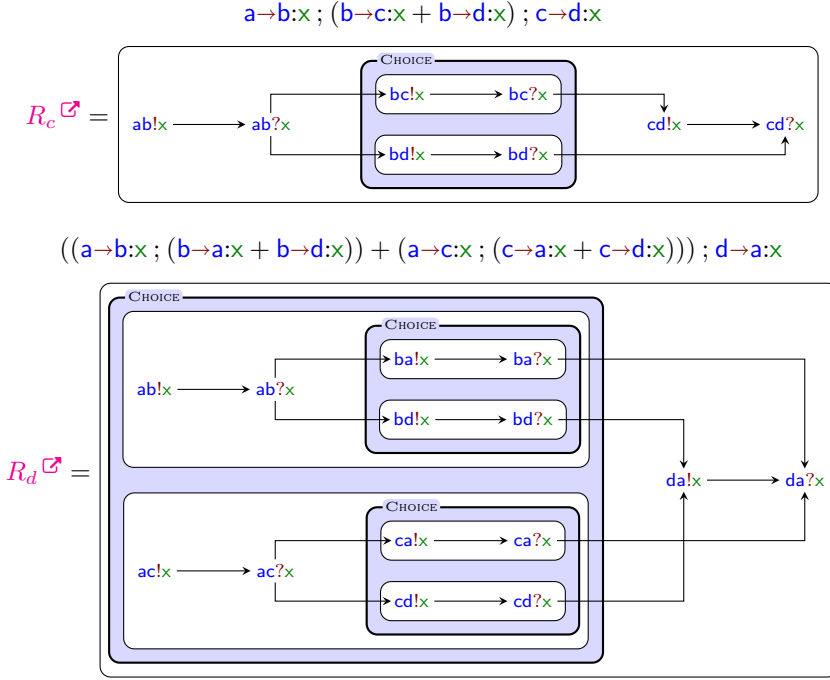


Figure 5.3: Two choreographies with the corresponding BPs.

this thesis is on supporting choices, we do not discuss this further and leave symbolic analyses for loops for future work.

Example 5.3. As an example, we construct the BP R_d in Figure 5.3: $\llbracket ((a \rightarrow b:x ; (b \rightarrow a:x + b \rightarrow d:x)) + (a \rightarrow c:x ; (c \rightarrow a:x + c \rightarrow d:x))) ; d \rightarrow a:x \rrbracket$. Let $\llbracket a \rightarrow b:x \rrbracket = (\{e_1, e_2\}, \preceq_1, \lambda_1, \mathcal{B}_1)$, $\llbracket b \rightarrow a:x \rrbracket = \dots$, $\llbracket b \rightarrow d:x \rrbracket = \dots$, $\llbracket a \rightarrow c:x \rrbracket = \dots$, $\llbracket c \rightarrow a:x \rrbracket = \dots$, $\llbracket c \rightarrow d:x \rrbracket = \dots$ and $\llbracket d \rightarrow a:x \rrbracket = (\{e_{13}, e_{14}\}, \preceq_7, \lambda_7, \mathcal{B}_7)$ as in Figure 5.4. First, $\llbracket b \rightarrow a:x + b \rightarrow d:x \rrbracket = (\{e_3, \dots, e_6\}, \preceq_2 \cup \preceq_3, \lambda_2 \cup \lambda_3, \{\{\mathcal{B}_2, \mathcal{B}_3\}\})$; this is the pairwise union of the first three components, with the branching structure adding a choice between the two branches. Then $\llbracket a \rightarrow b:x ; (b \rightarrow a:x + b \rightarrow d:x) \rrbracket = (\{e_1, \dots, e_6\}, \preceq_1 \cup \preceq_2 \cup \preceq_3 \cup \{(e_1, e_4), (e_2, e_3), (e_2, e_5)\}, \lambda_1 \cup \lambda_2 \cup \lambda_3, \mathcal{B}_1 \cup \{\{\mathcal{B}_2, \mathcal{B}_3\}\})$; this is the pairwise union of all components, with the addition of three dependencies: $e_2 \preceq e_3$ and $e_2 \preceq e_5$ represent the arrows in Figure 5.3 from $ab?x$ to $ba!x$ and $bd!x$ respectively as they all have subject b , while

$$\begin{aligned}
\llbracket \mathbf{1} \rrbracket &= (\emptyset, \emptyset, \emptyset, \emptyset) \\
\llbracket \mathbf{a} \rightarrow \mathbf{b} : \mathbf{x} \rrbracket &= (\{e_1, e_2\}, \{(e_1, e_1), (e_1, e_2), (e_2, e_2)\}, \{e_1 \mapsto \mathbf{ab}! \mathbf{x}, e_2 \mapsto \mathbf{ab}? \mathbf{x}\}, \{e_1, e_2\}) \\
\llbracket \mathbf{ab}? \mathbf{x} \rrbracket &= (\{e\}, \{(e, e)\}, \{e \mapsto \mathbf{ab}? \mathbf{x}\}, \{e\}) \\
\llbracket c_1 \dagger c_2 \rrbracket &= \llbracket c_1 \rrbracket \dagger \llbracket c_2 \rrbracket \text{ for } \dagger \in \{;, +, \parallel\} \\
\llbracket c^* \rrbracket &= \llbracket (c; c^*) + \mathbf{1} \rrbracket
\end{aligned}$$

In the following, let $R_i = (E_i, \preceq_i, \lambda_i, \mathcal{B}_i)$ for $i \in \{1, 2\}$ and let $E_i^{\mathbf{a}}$ be the subset of events in E_i with subject \mathbf{a} .

$$\begin{aligned}
R_1 \parallel R_2 &= (E_1 \cup E_2, \preceq_1 \cup \preceq_2, \lambda_1 \cup \lambda_2, \mathcal{B}_1 \cup \mathcal{B}_2) \\
R_1 ; R_2 &= (E_1 \cup E_2, \preceq_1 \cup \preceq_2 \cup \bigcup_{\mathbf{a} \in \mathcal{A}} E_1^{\mathbf{a}} \times E_2^{\mathbf{a}}, \lambda_1 \cup \lambda_2, \mathcal{B}_1 \cup \mathcal{B}_2) \\
R_1 + R_2 &= (E_1 \cup E_2, \preceq_1 \cup \preceq_2, \lambda_1 \cup \lambda_2, \{\{\mathcal{B}_1, \mathcal{B}_2\}\})
\end{aligned}$$

Figure 5.4: BP encoding of choreographies.

$e_1 \preceq e_4$ adds a direct dependency between $\mathbf{ab}! \mathbf{x}$ and $\mathbf{ba}? \mathbf{x}$ as they both have subject \mathbf{a} . The lower branch of the outer choice is analogous, and the choice between the two branches then consists of the pairwise union of the first three components, and the branching structure $\mathcal{B}_c = \{\{\mathcal{B}_u, \mathcal{B}_\ell\}\}$, where \mathcal{B}_u and \mathcal{B}_ℓ are the branching structures for the upper and lower branches respectively. Finally, the sequential composition with $\mathbf{d} \rightarrow \mathbf{a} : \mathbf{x}$ adds the events e_{13} and e_{14} , their dependencies and labels, the dependencies corresponding to the four additional arrows in Figure 5.3 and two additional ones from $\mathbf{ab}! \mathbf{x}$ and $\mathbf{ac}! \mathbf{x}$ to $\mathbf{da}? \mathbf{x}$, and the branching structure $\{e_{13}, e_{14}, \mathcal{B}_c\}$.

Remark (expressiveness). Not all BPs are obtainable from the choreography language in this chapter. As an example, consider the BP R_e in Figure 5.5, which mirrors the state machine in Figure 1.7: Alice (\mathbf{a}) and Bob (\mathbf{b}) both send the other a vote (\mathbf{v}), but they must send their own vote before receiving the other's. From our choreography language we can obtain BPs for $\mathbf{a} \rightarrow \mathbf{b} : \mathbf{v}$ and $\mathbf{b} \rightarrow \mathbf{a} : \mathbf{v}$, but we have no compositional operator to compose them in the desired way: as in the subsequent discussion in Chapter 1, parallel composition adds no dependencies at all, and sequential composition will also enforce an ordering on the send and receive events.

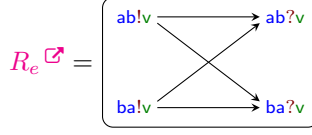


Figure 5.5: A BP representing a distributed vote with two voters.

5.4 Equivalence of BP encoding

For any given choreography c we can now derive two labelled transition systems: one from the operational semantics in Figure 5.2 over c , and one from the BP semantics in Figure 4.2 over the BP $\llbracket c \rrbracket$ produced by the rules in Figure 5.4. In the remainder of this section we show that, if c is dependently guarded, then the two transition systems are bisimilar, i.e., that $c \sim \llbracket c \rrbracket$. Recall from Chapter 2 that two systems are bisimilar if they are behaviourally indistinguishable, which is a stronger condition than language equivalence.

Example 5.4.

- $a \rightarrow b:x ; (b \rightarrow a:x + b \rightarrow a:y)$ is language equivalent but not bisimilar to $(a \rightarrow b:x ; b \rightarrow a:x) + (a \rightarrow b:x ; b \rightarrow a:y)$. In the former the choice between $b \rightarrow a:x$ and $b \rightarrow a:y$ is made only after $a \rightarrow b:x$, while in the latter the choice is made up front. As a result, it is possible in the latter system to fire $ab!x$; $ab?x$ and then end up in a state where $ba!x$ cannot be fired because the branch with $b \rightarrow a:y$ was chosen — or the other way around; in the former system it is always possible to fire both $ba!x$ and $ba!y$.
- $a \rightarrow b:x$ is bisimilar to $a \rightarrow b:x + a \rightarrow b:x$. While the latter contains a choice, the two systems cannot be distinguished by their behaviour. In both cases, the only allowed action is $ab!x$ and then $ab?x$.

Recall from Chapter 2 that, formally, $c \sim \llbracket c \rrbracket$ if there exists a relation \mathcal{R} such that $(c, \llbracket c \rrbracket) \in \mathcal{R}$ and, for every pair $(p, q) \in \mathcal{R}$:

- If $p \xrightarrow{\sigma} p'$ then $q \xrightarrow{\sigma} q'$ and $(p', q') \in \mathcal{R}$ for some q' , and vice-versa.
- If $p \downarrow$ then $q \downarrow$, and vice-versa.

This is also the approach we follow when proving that $c \sim \llbracket c \rrbracket$ for all (dependently guarded) choreographies c : we define a relation $\mathcal{R} = \{(c, \llbracket c \rrbracket) \mid c \text{ is a dependently guarded choreography}\}$ relating all dependently guarded

choreographies with their interpretation as BP by the rules in [Figure 5.4](#). We then show that:

- If $c \xrightarrow{\ell} c'$ then $\llbracket c \rrbracket \xrightarrow{\ell} \llbracket c' \rrbracket$ ([Lemma 5.6](#)).
- If $\llbracket c \rrbracket \xrightarrow{\ell} R'$ then $c \xrightarrow{\ell} c'$ such that $R' = \llbracket c' \rrbracket$ ([Lemma 5.7](#)).
- If $c \downarrow$ then $\llbracket c \rrbracket \downarrow$ ([Lemma 5.8](#)).
- If $\llbracket c \rrbracket \downarrow$ then $c \downarrow$ ([Lemma 5.9](#)).

Together these lemmas prove that $c \sim \llbracket c \rrbracket$ for all dependently guarded c ([Theorem 5.10](#)). Most of the proofs are straightforward by structural induction on c . Of particular interest, however, are the two reduction lemmas in the case of weak sequential composition, i.e., if $c_1 ; c_2 \xrightarrow{\ell} c'_1 ; c'_2$ in [Lemma 5.6](#) and if $\llbracket c_1 ; c_2 \rrbracket \xrightarrow{e} R'$ where e is an event in $\llbracket c_2 \rrbracket$ in [Lemma 5.7](#). To prove these specific cases we need to show a correspondence between partial termination and enabling events. We do this with [Lemma 5.5](#), in which we show two directions simultaneously. If the choreography c_1 can partially terminate for an action ℓ in c_2 then the BP $\llbracket c_1 ; c_2 \rrbracket$ can enable the corresponding event. Conversely, if $\llbracket c_1 ; c_2 \rrbracket$ can enable some event in $\llbracket c_2 \rrbracket$ then the choreography c_1 can partially terminate for its label. When proving these cases in [Lemmas 5.6](#) and [5.7](#), we then only have to show that the preconditions of [Lemma 5.5](#) hold.

Lemma 5.5. *Let c_1 and c_2 be dependently guarded choreographies. Let $c_2 \xrightarrow{\ell} c'_2$ and $\llbracket c_2 \rrbracket \xrightarrow{\check{e}} R'_2$ such that $\lambda(e) = \ell$ and $\llbracket c'_2 \rrbracket = R'_2 - e$.*

- (a) *If $c_1 \xrightarrow{\check{\ell}} c'_1$ then $\llbracket c_1 ; c_2 \rrbracket \xrightarrow{\check{e}} \llbracket c'_1 \rrbracket ; R'_2$.*
- (b) *If $\llbracket c_1 ; c_2 \rrbracket \xrightarrow{\check{e}} R'_1 ; R'_2$ then $c_1 \xrightarrow{\check{\lambda}(e)} c'_1$ and $\llbracket c'_1 \rrbracket = R'_1$.*

Proof sketch. This proof is by structural induction on c_1 . Although the details require careful consideration, it is conceptually straightforward: every case in (a) consists of showing that e is minimal and active in $\llbracket c'_1 \rrbracket ; R'_2$ and that $\llbracket c'_1 \rrbracket ; R'_2$ is the first refinement for which this is true, and then applying the second rule in [Figure 4.2b](#); every case in (b) consists of showing that $\llbracket c_3 ; c_2 \rrbracket \xrightarrow{\check{e}} \llbracket c'_3 \rrbracket ; R'_2$, or, depending on R'_1 , that $\llbracket c_3 ; c_2 \rrbracket \not\xrightarrow{\check{e}} \llbracket c'_3 \rrbracket ; R'_2$, for all relevant subexpressions c_3 of c_1 . We then apply the induction hypotheses to obtain $c_3 \xrightarrow{\check{\ell}} c'_3$ or $c_3 \not\xrightarrow{\check{\ell}}$ and, finally, apply the partial termination rules in [Figure 5.2c](#). \square

Lemma 5.6. *Let c be a dependently guarded choreography. If $c \xrightarrow{\ell} c'$ then $\llbracket c \rrbracket \xrightarrow{\ell} \llbracket c' \rrbracket$.*

Proof sketch. This proof is by structural induction on c . We note that, if $c = c_1 ; c_2$ and $c' = c'_1 ; c'_2$, i.e., when partial termination is applied, then the premises of [Lemma 5.5](#) hold by the induction hypothesis and the result swiftly follows. All other cases are straightforward. \square

Lemma 5.7. *Let c be a dependently guarded choreography. If $\llbracket c \rrbracket \xrightarrow{\ell} R'$ for some R' then $c \xrightarrow{\ell} c'$ such that $R' = \llbracket c' \rrbracket$.*

Proof sketch. This proof is by structural induction on c . We highlight two cases:

- If $c = c_1^*$ then we use a technical lemma to show that $R' = R'_1 ; \llbracket c_1^* \rrbracket$ such that $\llbracket c_1 \rrbracket \xrightarrow{\ell} R'_1$. It then follows from the induction hypothesis that $c_1 \xrightarrow{\ell} c'_1$ such that $\llbracket c'_1 \rrbracket = R'_1$. The remainder is straightforward.
- If $c = c_1 ; c_2$ then $\llbracket c \rrbracket = \llbracket c_1 \rrbracket ; \llbracket c_2 \rrbracket$. If e is an event in $\llbracket c_2 \rrbracket$ then we proceed to show that $\llbracket c_2 \rrbracket \xrightarrow{\ell} R'_2$, at which point we can apply the induction hypothesis. We have then satisfied the premises of [Lemma 5.5](#). The remainder is straightforward.

All other cases are straightforward. \square

Lemma 5.8. *Let c be a dependently guarded choreography. If $c \downarrow$ then $\llbracket c \rrbracket \downarrow$.*

Lemma 5.9. *Let c be a dependently guarded choreography. If $\llbracket c \rrbracket \downarrow$ then $c \downarrow$.*

The proofs for [Lemmas 5.8](#) and [5.9](#) are both by structural induction on c . All cases are straightforward.

Theorem 5.10. *Let c be a dependently guarded choreography. Then $c \sim \llbracket c \rrbracket$.*

Proof. Recall $\mathcal{R} = \{(c, \llbracket c \rrbracket) \mid c \text{ is a dependently guarded choreography}\}$. Let $R = \llbracket c \rrbracket$. Then:

- If $c \xrightarrow{\ell} c'$ then $R \xrightarrow{\ell} R'$ and $(c', R') \in \mathcal{R}$ (Lemma 5.6).
- If $R \xrightarrow{\ell} R'$ then $c \xrightarrow{\ell} c'$ and $(c', R') \in \mathcal{R}$ (Lemma 5.7).
- If $c \downarrow$ then $R \downarrow$ (Lemma 5.8).
- If $R \downarrow$ then $c \downarrow$ (Lemma 5.9).

Then \mathcal{R} is a bisimulation relation and, since (by definition) $(c, \llbracket c \rrbracket) \in \mathcal{R}$, we conclude that $c \sim \llbracket c \rrbracket$. \square

5.5 Conclusion

This chapter defined a choreography language and its operational semantics (Figures 5.1 and 5.2, respectively), using the weak sequential composition and partial termination of Rensink and Wehrheim [RW01], which is novel in the context of choreographies. We also showed that we can use BPs, as presented in Chapter 4, to model choreographies (Figure 5.4), and that a BP modelling a choreography is behaviourally equivalent (bisimilar) to the choreography's operational semantics (Theorem 5.10). Together with Chapter 4, this answers Question 2: combinations of concurrency and choice in choreographies can be compactly modelled as BPs.

A straightforward path for future work would be to investigate encodings of existing choreography languages as BPs, in particular global types in multiparty session types [HYC08].

Realisability of branching pomsets

This chapter is based on our FACS 2022 paper [FACS22] and on section 5 of the subsequent JLAMP 2024 paper [JLAMP24]. It also contains part of the conclusions and related work of these papers. It answers [Question 3](#). As in the original papers, we omit a number of technical lemmas and most of the proofs in favour of informal proof sketches or highlights. The omitted proofs and lemmas can be found in the technical report of our FACS 2022 paper [FACS22 TR].

6.1 Introduction

We have introduced the concept of realisability in [Chapter 1](#), the branching pomset (BP) model in [Chapter 4](#), and how to encode choreographies as BPs in [Chapter 5](#). In this chapter we formally define the realisability property for BPs with send and receive actions. We note that, while [Chapter 5](#) specifically encoded choreographies as BPs, the analysis in this chapter is solely based on BPs, regardless of their origin. As such, it is not restricted to BPs obtained from choreographies, but is more broadly applicable to BPs representing communication protocols — although part of the analysis consists of checking whether the BP has choreographic properties.

Our analysis draws inspiration from multiparty session types (MPST) [HYC08]. Through its syntax and projection operator, MPST defines a number of well-formedness conditions on global types which ensure realisability. We define

similar well-formedness conditions on BPs and prove that they ensure realisability as well. These conditions are sufficient but not necessary, i.e., a protocol may be realisable without being well-formed. We discuss some possible relaxations of the conditions at the end of the chapter.

We use the same set of labels \mathcal{L} for BPs as defined in [Chapter 5](#): send and receive actions $\mathbf{ab!x}$ and $\mathbf{ab?x}$, ranged over by ℓ , where $\mathbf{ab!x}$ is a send from \mathbf{a} to \mathbf{b} of a message of type \mathbf{x} and $\mathbf{ab?x}$ is the corresponding receive by \mathbf{b} .

6.2 Realisability

We model distributed implementations as compositions of a collection of ‘local’ BPs \vec{R} and ordered buffers (FIFO queues) \vec{b} containing the messages in transit (sent but not yet received) between directed pairs of processes (or channels), similar to communicating finite-state machines [\[BZ83\]](#). A BP is *local* if it only contains actions for a single process; a composition should contain one local BP for each process and one buffer for each channel.

Composition is formally defined below. We use three auxiliary functions: $add(\mathbf{ab:x}, \vec{b})$ returns \vec{b} with \mathbf{x} added to $b_{\mathbf{ab}}$, $remove(\mathbf{ab:x}, \vec{b})$ returns \vec{b} with \mathbf{x} removed from $b_{\mathbf{ab}}$, and $has(\mathbf{ab:x}, \vec{b})$ returns whether \mathbf{x} is pending in $b_{\mathbf{ab}}$. Since we consider ordered buffers, add appends message types to the end of the corresponding queue, $remove$ removes message types from the front, and has only checks whether the first message matches. Formally:

$$\begin{aligned} add(\mathbf{ab:x}, \vec{b}) &= \vec{b}[b_{\mathbf{ab}} \cdot \mathbf{x} / b_{\mathbf{ab}}] \\ remove(\mathbf{ab:x}, \vec{b}) &= \vec{b}[t / b_{\mathbf{ab}}] \text{ where } b_{\mathbf{ab}} = \mathbf{x} \cdot t \\ has(\mathbf{ab:x}, \vec{b}) &\Leftrightarrow b_{\mathbf{ab}} = \mathbf{x} \cdot t \text{ for some } t \in \mathcal{X}^* \end{aligned}$$

We note that our termination condition does not require the buffers to be empty. In practice asynchronous communication channels will typically have some latency, and requiring empty buffers would require processes (the local BPs) to be aware of messages in transit. Instead, in our model the presence or absence of orphan messages (messages unreceived on termination) is a separate property from realisability, to be verified in isolation. It does, however, follow from our well-formedness conditions in [Section 6.3](#) that a well-formed and realisable protocol is also free of orphan messages.

Definition 6.1 (Composition). Let \vec{R} be a process-indexed vector of local BPs. Let \vec{b} be a channel-indexed vector of ordered buffers. Their composition is the tuple (\vec{R}, \vec{b}) , whose semantics is defined as the labeled transition system defined by the rules below.

$$\frac{R_a \xrightarrow{ab!x} R'_a \quad \vec{b}' = \text{add}(ab:x, \vec{b})}{(\vec{R}, \vec{b}) \xrightarrow{ab!x} (\vec{R}[R'_a/R_a], \vec{b}')} [\text{SEND}] \quad \frac{R_b \xrightarrow{ab?x} R'_b \quad \text{has}(ab:x, \vec{b}) \quad \vec{b}' = \text{remove}(ab:x, \vec{b})}{(\vec{R}, \vec{b}) \xrightarrow{ab?x} (\vec{R}[R'_b/R_b], \vec{b}')} [\text{RECEIVE}]$$

$$\frac{\forall a : R_a \downarrow}{(\vec{R}, \vec{b}) \downarrow} [\text{TERMINATE}]$$

A protocol is *realisable* if there exists a faithful distributed implementation of it, i.e., one defining the same behaviour. We formally define realisability below. We note that it is typically defined in terms of language (trace) equivalence [GT19]. However, as the exact branching of choices plays an important part in BPs, we use a more strict notion of equivalence and require the global BPs and the composition to be bisimilar, as in Chapter 5. We note that our well-formedness conditions enforce a deterministic setting, in which bisimilarity agrees with language equivalence. We then choose to prove bisimilarity rather than language equivalence because the proofs are typically more straightforward.

Recall from Chapter 2 that two BPs R_1, R_2 are bisimilar, written $R_1 \sim R_2$, if (a) for every reduction $R_1 \xrightarrow{\ell} R'_1$ there exists a reduction $R_2 \xrightarrow{\ell} R'_2$ such that R'_1 and R'_2 are again bisimilar, and vice-versa, and (b) R_1 can terminate iff R_2 can.

Definition 6.2 (Realisability). Let R be a BP. The protocol it represents is realisable if there exists a composition (\vec{R}, \vec{b}) such that b_{ab} is empty for all ab and $R \sim (\vec{R}, \vec{b})$.

Example 6.3. Consider the BPs in Figure 6.1:

- R_f is unrealisable. Alice and Bob both have to send a **yes** or a **no** to the other but the two messages must be the same. It is impossible without further synchronisation or communication to prevent a scenario in which one will send a different message than the other.

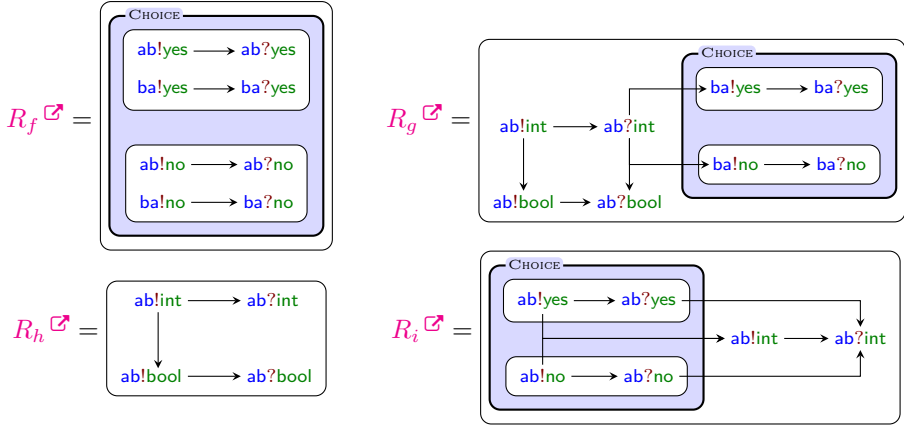


Figure 6.1: A collection of realisable and unrealisable BPs.

- R_g is realisable. Alice first sends an **int** and then a **bool** to Bob. After receiving the **int**, Bob returns either a **yes** or a **no**.
- R_h is unrealisable. Alice sends an **int** and a **bool** to Bob, but while they agree that Alice first sends the **int** and then the **bool**, the order in which Bob receives the message is unspecified. As we assume ordered buffers, Bob will first receive the **int**, but the global BP allows an execution in which Bob first receives the **bool**.
- R_i is realisable. Alice sends a **yes** or a **no** to Bob, followed by an **int**.

We note that it is easy to go from a global BP R to a local BP for some process a by *projecting* it on a , written $R|_a$. We will use projections in our well-formedness conditions and realisability proof, and we formally define them below. The projection $R|_a$ restricts R to the events whose subject is a , and restricts \preceq and λ accordingly. The branching structure is pruned by removing all discarded events (leaves), but no inner nodes of the tree are removed, even if they are left without any children. This is done to safeguard the symmetry with the branching structure of R to ease our proofs.

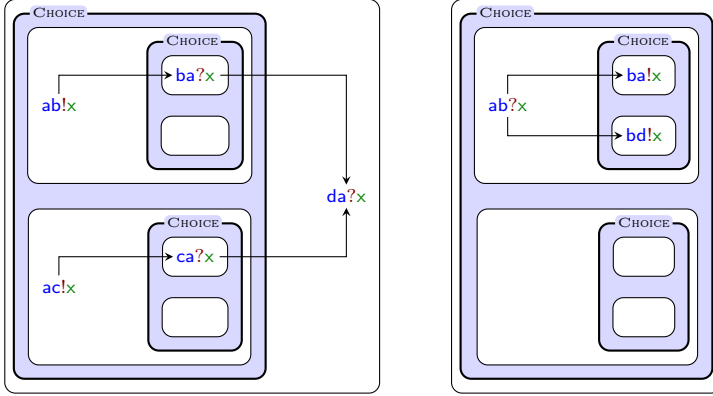


Figure 6.2: The projections of the BP R_d in Figure 5.3 on **a** (left) and **b** (right).

Definition 6.4 (Projection). $(E, \preceq, \lambda, \mathcal{B})|_a = (E_a, \preceq_a, \lambda_a, \mathcal{B}_a)$ where:

- $E_a = \{e \in E \mid \text{subj}(\lambda(e)) = \mathbf{a}\}$
- $\preceq_a = \preceq \cap (E_a \times E_a)$
- $\lambda_a = \lambda \cap (E_a \times \mathcal{L})$
- $\mathcal{B}_a = \mathcal{B}|_a$ as defined below.

$$\begin{aligned}
 e|_a &= e \text{ if } e \in E_a \\
 \{\mathcal{C}_1, \dots, \mathcal{C}_n\}|_a &= \{\mathcal{C}_i|_a \mid 1 \leq i \leq n \wedge \mathcal{C}_i|_a \text{ is defined}\} \\
 \{\mathcal{B}_1, \mathcal{B}_2\}|_a &= \{\mathcal{B}_1|_a, \mathcal{B}_2|_a\}
 \end{aligned}$$

As an example, Figure 6.2 shows the projections of the BP R_d in Figure 5.3 (page 95) on **a** and **b**. The events with different subjects are removed, as are dependencies involving them. All choices and branches remain, even if both branches of a choice are empty, as is the case for a choice in the projection on **b**. We note that, as the graphical representation of a BP shows the transitive reduction of the causality relation and not the full relation, it is unclear from just Figure 5.3 whether, for example, the projection on **a** should contain a dependency between $ab!x$ and $ba?x$. However, this is unambiguous in the choreography included in Figure 5.3, from which R_d is derived, and it follows from the encoding (Figure 5.4) that the events are related.

Finally, we prove that R and its projections can mirror each other's refinements. Both proofs are straightforward by induction on the structure of the premise's derivation tree.

Lemma 6.5. *If $R \sqsupseteq R'$ then $R|_a \sqsupseteq R'|_a$.*

Lemma 6.6. *If $R|_a \sqsupseteq R'_a$ then $R \sqsupseteq R'$ for some R' such that $R'_a = R'|_a$.*

6.3 Well-formedness

We define four well-formedness conditions on BPs: to be well-formed, a BP must be well-branched, well-channeled, tree-like and choreographic. Well-branchedness, well-channeledness and tree-likeness are inspired by MPST [HYC08] and ensure some safety properties. Choreographicness ensures that the BP represents some sort of meaningful protocol.

- **Well-branched** (Definition 6.8): every choice is made only on the label of a send-receive pair, i.e., the first events in every branch must be a send and receive, respectively, between some processes a and b , with the message type being different in every branch. Additionally, the projection on every process uninvolved in the choice must be the same in every branch. Then a and b are both aware of the chosen branch and all other processes are unaffected by the choice.

Although the BP model only contains binary choices, an n -ary choice \mathcal{C} can be encoded as a nested binary one, where the n children of \mathcal{C} become the leaves of a binary tree. We call such a leaf \mathcal{B} an *option* of \mathcal{C} , written $\mathcal{B} \text{ opt } \mathcal{C}$, which is formally defined below. This allows us to properly interpret \mathcal{C} as an n -ary choice again and reason about it accordingly.

- **Well-channeled** (Definition 6.9): pairs of sends and pairs of receives on the same channel that can occur in the same execution should be ordered, and the pairs of sends should have the same order as the pairs of their corresponding receives. A BP which is not well-channeled could, for example, yield a trace $ab!x; ab!y; ab?y; ab?x$, which cannot be reproduced by a composition using ordered buffers.
- **Tree-like** (Definition 6.10): events inside of choices can only affect future events in the same branch. Graphically speaking, arrows can only enter boxes, not leave them. As a consequence, the causality relation \preceq follows

the branching structure \mathcal{B} and has a tree-like shape — hence the name.

- **Choreographic** (Definition 6.11): the BP represents a choreography of some sort, i.e., a communication protocol in which the send and receive events are properly paired and all dependencies can be logically derived. Specifically, all dependencies are between send-receive pairs or between same-subject events, or they can be transitively derived from those. Additionally, there is the following correspondence between the send and receive events: every send can be matched to exactly one corresponding receive, and every non-top-level receive has some corresponding send at the same level of the branching structure \mathcal{B} . This definition is similar to the definition of well-formedness by Guanciale and Tuosto [GT19].

Definition 6.7 (Option). Let $\mathcal{B}, \mathcal{C} \succ R.\mathcal{B}$. Then \mathcal{B} is an *option* of \mathcal{C} , written $\mathcal{B} \text{ opt } \mathcal{C}$, if $\mathcal{B} \in \{\mathcal{B}^\dagger \mid \mathcal{B}^\dagger \text{ opt}^\dagger \mathcal{C} \wedge \nexists \mathcal{B}^\ddagger : (\mathcal{B}^\ddagger \text{ opt}^\dagger \mathcal{C} \wedge \mathcal{B}^\ddagger \succ \mathcal{B}^\dagger)\}$, where opt^\dagger is defined as follows:

$$\frac{\mathcal{B} \in \mathcal{C}}{\mathcal{B} \text{ opt}^\dagger \mathcal{C}} \quad \frac{\mathcal{B} \in \mathcal{C}' \quad \{\mathcal{C}'\} \text{ opt}^\dagger \mathcal{C}}{\mathcal{B} \text{ opt}^\dagger \mathcal{C}}$$

Definition 6.8 (Well-branched). A BP R is *well-branched* if, for every $\mathcal{C} \succ R.\mathcal{B}$ there exist processes \mathbf{a}, \mathbf{b} such that for every $\mathcal{B}_i \neq \mathcal{B}_j \text{ opt } \mathcal{C}$ there exist events $e_{i1}, e_{i2} \in \mathcal{B}_i$ and $e_{j1}, e_{j2} \in \mathcal{B}_j$ such that:

- $\lambda(e_{i1}) = \mathbf{ab}!x$, $\lambda(e_{i2}) = \mathbf{ab}?x$, $\lambda(e_{j1}) = \mathbf{ab}!y$ and $\lambda(e_{j2}) = \mathbf{ab}?y$ for some $x \neq y$;
- $e_{i1} \preceq e_i$ for all $e_i \succeq \mathcal{B}_i$ and $e_{j1} \preceq e_j$ for all $e_j \succeq \mathcal{B}_j$;
- $e_{i2} \preceq e_i$ for all $e_i \succeq \mathcal{B}_i$ for which $\text{subj}(e_i) = \mathbf{b}$ and $e_{j2} \preceq e_j$ for all $e_j \succeq \mathcal{B}_j$ for which $\text{subj}(e_j) = \mathbf{b}$; and
- $R|_{\mathcal{B}_i} \downarrow_{\mathbf{c}} = R|_{\mathcal{B}_j} \downarrow_{\mathbf{c}}$ for all $\mathbf{c} \neq \mathbf{a}, \mathbf{b}$.²¹

Definition 6.9 (Well-channeled). A BP R is *well-channeled* if, for all events $e_1, e_2, e_3, e_4 \in R.E$:

²¹Technically $R|_{\mathcal{B}_i} \downarrow_{\mathbf{c}}$ and $R|_{\mathcal{B}_j} \downarrow_{\mathbf{c}}$ have different events and should thus be isomorphic rather than precisely equal. We choose to write it as an equality to not unnecessarily complicate the definition and proofs.

- If e_1 and e_2 are either both sends or both receive events, and if they share the same channel, then they are either causally ordered or mutually exclusive.
- If e_1, e_3 and e_2, e_4 are two pairs of matching send-receive events sharing the same channel, and if there exists no $e_5 \in R.E$ such that $e_1 \prec e_5 \prec e_3$ or $e_2 \prec e_5 \prec e_4$, then $e_1 \preceq e_2 \implies e_3 \preceq e_4$.

Definition 6.10 (Tree-like). A BP R is *tree-like* if:

$$\forall \mathcal{C} = \{\mathcal{B}_1, \mathcal{B}_2\} \succ R.\mathcal{B} : (e_1 \preceq e_2 \wedge e_1 \succeq \mathcal{B}_i) \implies e_2 \succeq \mathcal{B}_i, \text{ where } i \in \{1, 2\}.$$

Definition 6.11 (Choreographic). A BP R is *choreographic* if, for every $e \in R.E$:

- If there exists $e' \in R.E$ such that $e' \prec e$ then there exists some event e'' (not necessarily distinct from e') such that $e' \preceq e'' \prec e$ and either $\text{subj}(\lambda(e'')) = \text{subj}(\lambda(e))$ or $[\lambda(e'') = \mathbf{ab!x}$ and $\lambda(e) = \mathbf{ab?x}$ for some $\mathbf{a}, \mathbf{b}, \mathbf{x}]$.
- If $\lambda(e) = \mathbf{ab?x}$ and $e \in \mathcal{B}$ for some $\mathcal{B} \succ R.\mathcal{B}$ then there exists some e' such that $e' \in \mathcal{B}$ and $\lambda(e') = \mathbf{ab!x}$ and $e' \prec e$.
- If $\lambda(e) = \mathbf{ab!x}$ then there exists exactly one e' such that $e \preceq e'$ and that $\lambda(e') = \mathbf{ab?x}$ and that $\forall e^\dagger : (\lambda(e^\dagger) = \mathbf{ab!x} \wedge e^\dagger \preceq e) \implies e^\dagger \preceq e$.

Definition 6.12 (Well-formed). A BP R is *well-formed* if it is well-branched, well-channeled, tree-like and choreographic.

Example 6.13. Recall the BPs in [Figure 6.1](#):

- R_f is not well-formed since it is not well-branched: for example, the branches of the choice have multiple minimal events. It is indeed unrealisable.
- R_g is both well-formed and realisable.
- R_h is not well-formed since it is not well-channeled: the two receive events are on the same channel but are unordered. It is indeed unrealisable.

- $R_i \curvearrowright$ is not well-formed since it is not tree-like: there are arrows from events inside branches of a choice to ab!int and ab?int , even though the latter are not part of the same branch. It is, however, realisable: by duplicating the events ab!int and ab?int and moving one copy inside each branch, we obtain an equivalent BP which is well-formed. This illustrates that, while we later prove that our well-formedness conditions are sufficient, they are not necessary.

Finally, we show that well-formedness is retained after a reduction.

Lemma 6.14. *Let R be a BP and let $R \xrightarrow{\ell} R'$. If R is well-formed then so is R' .*

Proof sketch. We use that the components of R' are subsets of, or derived from (in the case of the branching structure), the components of R . It then follows that a violation of one of the well-formedness conditions in R' would also invariably lead to a violation of one of the conditions in R . \square

6.4 Realisability proof

To prove that a BP R 's protocol is realisable, we must show the existence of a bisimilar composition of local BPs and buffers. We thus define a canonical decomposition of R by combining projections with a buffer construction. If R is well-formed, we prove that this decomposition is bisimilar to R . The (re)construction of the buffer contents of channel ab based on R , written $\text{buff}_{\text{ab}}(R)$, and the canonical decomposition of R , written $\text{cd}(R)$, are defined below.

The buffer construction $\text{buff}_{\text{ab}}(R)$ gathers the receive events in R that have no preceding matching send event. We infer that, since the send has already been fired and the receive has not, the message must be in transit.

Definition 6.15 (Buffer construction). Let R be a BP. Let \mathbf{a} and \mathbf{b} be processes in R . Let ε be the empty word.

$$\text{Then } \text{buff}_{\text{ab}}(R) = \begin{cases} \mathbf{x} \cdot \text{buff}_{\text{ab}}(R') & \text{if } R' = R - e \text{ and } \lambda(e) = \text{ab?x} \\ & \text{and } \forall e': \text{if } e' \prec e \text{ then } \lambda(e') \neq \text{ab!x} \\ & \text{and } \forall e', \mathbf{y}: \text{if } e' \prec e \text{ then } \lambda(e') \neq \text{ab?y} \\ \varepsilon & \text{otherwise.} \end{cases}$$

The corresponding message types are nondeterministically put in some order that respects the order of the gathered receive events — if $e_1 \prec e_2$ then e_1 's message type must precede that of e_2 in the constructed buffer. We note that all unmatched receive events are top-level if R is choreographic, and that the same-channel top-level receive events are totally ordered if R is well-channeled. It follows that, although it may add duplicate messages and is nondeterministic in the general case, $\text{buff}_{\text{ab}}(R)$ does not add duplicate messages and is deterministic, i.e., it is uniquely defined, if R is well-formed — or, more specifically, if R is at least well-channeled and choreographic.

Definition 6.16 (Canonical decomposition). Let R be a BP. Let \vec{R} be such that $R_{\mathbf{a}} = R|_{\mathbf{a}}$ for all \mathbf{a} . Let \vec{b} be such that $b_{\mathbf{ab}} = \text{buff}_{\mathbf{ab}}(R)$ for all \mathbf{ab} . Then $cd(R) = (\vec{R}, \vec{b})$ is the *canonical decomposition* of R .

To prove that a well-formed R is bisimilar to $cd(R)$, we define the relation $\mathcal{R} = \{(R, (\vec{R}^\dagger, \vec{b})) \mid (\vec{R}^\dagger, \vec{b}) \sim (\vec{R}, \vec{b}) = cd(R)\}$ and we prove that \mathcal{R} is a bisimulation relation ([Theorem 6.24](#)). Note that the vector of buffers \vec{b} is the same across the definition; we only allow leeway in the vector of local BPs. The proof consists of the two parts mentioned in [Section 6.2](#). Given that $(R, (\vec{R}^\dagger, \vec{b})) \in \mathcal{R}$, if one can make some reduction then the other must be able to make the same reduction such that the resulting configurations are again related by \mathcal{R} ([Lemma 6.20](#), [Lemma 6.21](#)). Additionally, if one can terminate then so should the other ([Lemma 6.22](#), [Lemma 6.23](#)).

The reason that \mathcal{R} is not simply the set of all $(R, cd(R))$ is that a reduction from $cd(R)$ may not always result in $cd(R')$ for some R' . This is because choices are only resolved in the BP of the process causing the reduction. For example, consider BP R_i in [Figure 6.1](#). Upon Alice sending **yes** the global BP would resolve the choice for both processes simultaneously. However, upon Alice sending **yes** in the canonical decomposition the projection on Bob remains unchanged and still contains receive events for both **yes** and **no**. Since **yes** has been added to the buffer from Alice to Bob, we know that Bob will eventually have to pick the branch containing **yes** — after all, there is no **no** to receive. In other words: this configuration is bisimilar to the canonical decomposition of the resulting global BP, in which the choice has also been resolved for Bob. If there were also some additional **no** being sent from Alice to Bob, e.g., if we replace the messages **int** in R_i with **no**, then R_i being well-channeled and the buffers being ordered would still ensure that we can safely resolve Bob's choice. This crucial insight is formally proven in [Lemma 6.17](#).

Lemma 6.17. *Let R be a well-formed BP. Let $(\vec{R}, \vec{b}) = cd(R)$. Let ℓ be some label and let $\mathbf{a} = \text{subj}(\ell)$. If $R \xrightarrow{\ell} R'$ and if $(\vec{R}, \vec{b}) \xrightarrow{\ell} (\vec{R}[R'_a/R|_a], \vec{b}^\dagger)$ and if $R'_a = R'|_a$, then $(\vec{R}[R'_a/R|_a], \vec{b}^\dagger) \sim (\vec{R}', \vec{b}') = cd(R')$.*

Proof sketch. If $\ell = \mathbf{ba}?\mathbf{x}$ for some \mathbf{b}, \mathbf{x} , then it follows from the well-formedness of R that $R' = R - e$ and the remainder is straightforward. The same is true if $\ell = \mathbf{ab}!\mathbf{x}$ and e is top-level, i.e., $e \in R.\mathcal{B}$.

Otherwise it follows from the well-formedness of R that e is a minimal send event in one of the options of a top-level choice, i.e., $e \in \mathcal{B} \text{ opt } \mathcal{C} \in R.\mathcal{B}$ for some \mathcal{B}, \mathcal{C} , and $R' = R|_{(R.\mathcal{B} \setminus \mathcal{C}) \cup (\mathcal{B} - e)}$. We show that the set of unmatched receive events in R' is exactly that of R with the addition of the one corresponding to e , and then $\vec{b}' = \text{add}(\mathbf{ab}:\mathbf{x}, \vec{b}) = \vec{b}^\dagger$. It follows that $(\vec{R}[R'_a/R|_a], \vec{b}^\dagger) = (\vec{R}[R'_a/R|_a], \vec{b}')$. For the projections, we proceed in two steps:

- First we observe that, since R is well-branched, $\mathcal{B}'|_c = \mathcal{B}|_c$ for all $\mathcal{B}' \text{ opt } \mathcal{C}$ and for all $c \neq \mathbf{a}, \mathbf{b}$. It follows that $R|_c \sim R'|_c$, and then $(\vec{R}[R'_a/R|_a], \vec{b}') \sim (\vec{R}'[R|_b/R'|_b], \vec{b}')$. Note that the projection on \mathbf{a} is $R'|_a$ and the projection on \mathbf{b} is $R|_b$ on both sides, and the projection on every other \mathbf{c} is bisimilar.
- Next we show that, with the new message added to the buffer, no event can ever fire in $R|_b$ in any other option of \mathcal{C} than \mathcal{B} . It follows that we can discard all other options, and then $(\vec{R}'[R|_b/R'|_b], \vec{b}') \sim (\vec{R}', \vec{b}') = cd(R')$. \square

To satisfy the preconditions of Lemma 6.17, we additionally prove that R 's reductions can be mirrored by its projection on the reduction label's subject (Lemma 6.18) and dually that the reductions of R 's canonical decomposition can be mirrored by R (Lemma 6.19). Their proofs rely on the observations that the corresponding event e must be minimal both in R and in $R|_a$, and that the branching structures of the two are the same (modulo discarded leaves). It then follows that the same refinement enables e in both R and $R|_a$.

Lemma 6.18. *Let R be a tree-like BP. If $R \xrightarrow{\ell} R'$ and $\mathbf{a} = \text{subj}(\ell)$ then $R|_a \xrightarrow{\ell} R'|_a$.*

Lemma 6.19. *Let R be a well-channeled, tree-like and choreographic BP. Let $(\vec{R}, \vec{b}) = cd(R)$. If $(\vec{R}, \vec{b}) \xrightarrow{\ell} (\vec{R}[R'_a/R]_a, \vec{b}')$ then $R \xrightarrow{\ell} R'$ for some R' such that $R'_a = R'_a$.*

Finally, we bring everything together and prove the four necessary steps for bisimulation in the lemmas below, culminating in [Theorem 6.24](#). The proof for [Lemma 6.20](#) uses [Lemma 6.18](#) to show the preconditions of [Lemma 6.17](#) and then applies the latter. This gives us $cd(R) \xrightarrow{\ell} cd(R)' \sim cd(R')$, and since $(\vec{R}^\dagger, \vec{b}^\dagger) \sim cd(R)$ the result is then straightforward. The proof for [Lemma 6.21](#) is analogous but uses [Lemma 6.19](#). The proofs for [Lemma 6.22](#) and [Lemma 6.23](#) respectively use [Lemma 6.5](#) and [Lemma 6.6](#) to show that, if one can terminate by refining to the empty set, then so must the other.

Lemma 6.20. *Let $(R, (\vec{R}^\dagger, \vec{b})) \in \mathcal{R}$. If R is well-formed and $R \xrightarrow{\ell} R'$ then there exist \vec{R}^\ddagger and \vec{b}^\ddagger such that $(\vec{R}^\dagger, \vec{b}) \xrightarrow{\ell} (\vec{R}^\ddagger, \vec{b}^\ddagger)$ and $(R', (\vec{R}^\ddagger, \vec{b}^\ddagger)) \in \mathcal{R}$.*

Lemma 6.21. *Let $(R, (\vec{R}^\dagger, \vec{b})) \in \mathcal{R}$. If R is well-formed and $(\vec{R}^\dagger, \vec{b}) \xrightarrow{\ell} (\vec{R}^\ddagger, \vec{b}^\ddagger)$ then there exists R' such that $R \xrightarrow{\ell} R'$ and $(R', (\vec{R}^\ddagger, \vec{b}^\ddagger)) \in \mathcal{R}$.*

Lemma 6.22. *Let $(R, (\vec{R}^\dagger, \vec{b})) \in \mathcal{R}$. If R is well-formed and $R \downarrow$ then $(\vec{R}^\dagger, \vec{b}) \downarrow$.*

Lemma 6.23. *Let $(R, (\vec{R}^\dagger, \vec{b})) \in \mathcal{R}$. If R is well-formed and $(\vec{R}^\dagger, \vec{b}) \downarrow$ then $R \downarrow$.*

Theorem 6.24. *Let R be a BP. If R is well-formed and $buff_{ab}(R) = \varepsilon$ for all ab then the protocol represented by R is realisable.*

Proof. It follows from [Lemmas 6.20](#) to [6.23](#) that \mathcal{R} is a bisimulation relation. Specifically, it then follows that $R \sim cd(R)$. Then, since $buff_{ab}(R) = \varepsilon$ for all ab , by [Definition 6.2](#) the protocol represented by R is realisable. \square

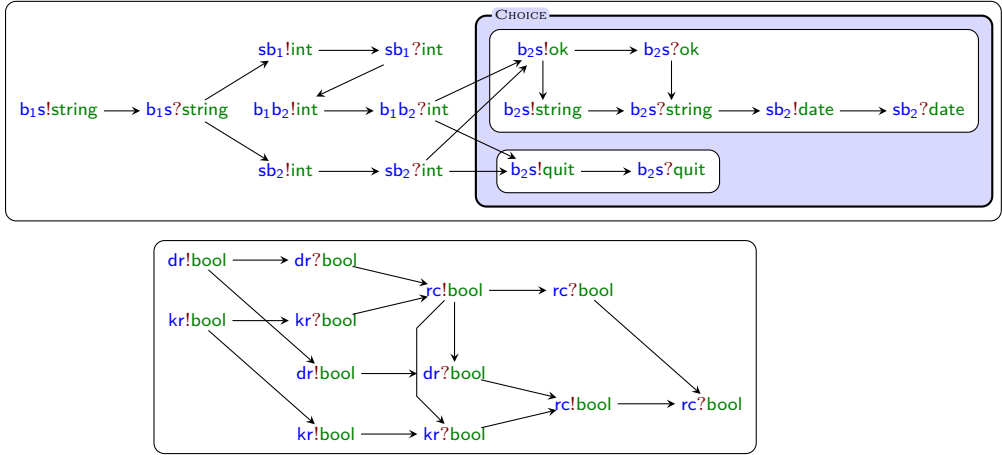


Figure 6.3: BPs representing the **two-buyers-protocol** (top) and two iterations of the **simple streaming protocol** (bottom) [HYC08].

6.5 Examples

We briefly look at two example protocols used by Honda et al. [HYC08]. Both are depicted as BPs in Figure 6.3.

The **two-buyers-protocol** (top) features Buyer 1 and Buyer 2 (b_1, b_2) who wish to jointly buy a book from Seller (s). Buyer 1 first sends the title of the book (**string**) to Seller, Seller sends its quote (**int**) to both Buyer 1 and Buyer 2, and Buyer 1 sends Buyer 2 the amount they can contribute (**int**). Buyer 2 then notifies Seller whether they accept (**ok**) or reject (**quit**) the quote. If they accept, they also send their address (**string**), and Seller returns a delivery date (**date**).

The **simple streaming protocol** (bottom) features Data Producer (d) and Key Producer (k), who continuously respectively send data and keys (both **bool**) to Kernel (r). Kernel performs some computation and sends the result (**bool**) to Consumer (c). The protocol in Figure 6.3 shows two iterations of this process.

Both BPs are well-formed, and hence the protocols are realisable. We note that, as in the paper by Honda et al., further communication between Buyers 1 and 2 has been omitted in the two-buyers-protocol. Since this is bound to be different in the case of acceptance and rejection, because Buyer 2 does not inform Buyer 1 about their decision, the resulting BP would not be well-branched

and thus not well-formed — though still realisable. We discuss relaxed well-branchedness conditions later in this chapter. Also note that the `ok` and address (`string`) messages are sent sequentially; `sending these in parallel` would violate well-channelledness and make the protocol unrealisable with ordered buffers.

The same is true for the streaming protocol: the two iterations are composed sequentially and `doing so concurrently` would violate well-channelledness and result in unrealisability. The size of the BP for the streaming protocol scales linearly with the number of depicted iterations; showing all (infinitely many) iterations would require an infinitely large BP. We have briefly touched upon infinity in [Chapter 4](#).

6.6 Related work

Realisability Realisability has been well-studied in the last two decades in a variety of settings. For example, Alur et al. study the realisability of message sequence charts [\[AEY03\]](#). They define the notions of weak and safe realisability of languages, the latter also ensuring deadlock-freedom, and they define closure conditions on languages which they show to precisely capture weakly and safely realisable languages. Lohmann and Wolf define the notion of distributed realisability, where a protocol is distributedly realisable if there exists a set of compositions such that every composition covers a subset of the protocol and the entire protocol is covered by their union [\[LW09\]](#). Fu et al. [\[FBS04\]](#), Basu et al. [\[BBO12\]](#), Finkel and Lozes [\[FL17\]](#) and Schewe et al. [\[SAB20\]](#) all study the realisability of protocols on different network configurations when considering only the sending behaviour — receive events are omitted — showing necessary and/or sufficient conditions and decidability results. Our definition of realisability closely resembles the one used by Montesi [\[Mon23\]](#). As noted by Montesi, there also exist settings which only require soundness, not completeness. In such a setting, a system is compliant if its behaviour is a subset of that specified by the choreography; it does not have to cover all of it.

Realisability of pomsets One major source of inspiration for our work has been the work by Guanciale and Tuosto, in which they presented a realisability analysis based on sets of pomsets [\[GT19\]](#). They show how to capture the language closure conditions of Alur et al. [\[AEY03\]](#) directly on pomsets, without having to explicitly compute their language. Typically choreography languages are limited in their expressiveness and any analysis on their realisability is then language-dependent. Both Alur et al. and Guanciale and Tuosto perform

a syntax-oblivious analysis, which has the benefit of being applicable to any specification which can be encoded as a set of pomsets, regardless of the specification language. The analysis by Guanciale and Tuosto is at a higher level of abstraction than sets of traces. This allows both for a more efficient analysis and for easier identification of design errors, as these can be identified in a more abstract model.

Our approach is similarly syntax-oblivious, though the analysis itself is based on multiparty session types. A major technical difference is that Guanciale and Tuosto use unordered buffers (e.g., non-FIFO queues) while ours are ordered. For example, the parallel composition of $a \rightarrow b : x$ and $a \rightarrow b : y$ is realisable in the unordered setting and not in the ordered one. The two settings agree on realisability when the two message types are the same (e.g., two concurrent copies of $a \rightarrow b : x$); while Guanciale and Tuosto explicitly note that they support concurrently repeated actions, however, our current well-channelledness condition does not make an exception for these. This is an obvious target for relaxation of our conditions.

Further inspiration for relaxation of well-formedness conditions can be found in an earlier paper by the same authors [TG18]. In particular their definition of well-branchedness, using ‘active’ and ‘passive’ agents, could serve as a basis for a relaxed version of our own.

A meaningful in-depth comparison between the pomset-based approach by Guanciale and Tuosto [GT19] and ours, both in terms of expressiveness and efficiency, would first require further development (relaxation) of our conditions and is therefore left for future work. In the meantime, one takeaway from their paper is the performance gain they obtain by lifting the analysis from languages (sets of traces) to a higher level of abstraction, i.e., sets of pomsets. It would be interesting to investigate whether a further performance gain can be obtained by lifting their analysis from sets of pomsets to a model such as branching pomsets or an appropriate class of event structures.

Multiparty session types The other major source of inspiration for our work is multiparty session types (MPST), introduced by Honda et al. [HYC08]:

- Our well-branchedness condition corresponds to the branching syntax of global types in MPST, as introduced in 2008, and its definition of projection. Branching in global types is done on the label of a single message, and the projection on agents uninvolved in the choice is only defined if it is the same in every branch.

- Our well-channeledness condition corresponds to the principle that same-channeled actions should be ordered. We note that our condition is more liberal: we prohibit concurrent sends or receives on the same channel, while in nearly all MPST systems the projection of a parallel composition on an agent is undefined if the agent occurs in both parallel branches (even if those branches' channels are disjoint); a paper by Jongmans and Ferreira is a very recent exception [JF23].
- Our tree-likeness condition follows from the syntax of global types in MPST, which uses a prefix operator rather than the more general sequential composition. As a consequence all global types are tree-like. The same is true for other languages that use a prefix operator and not sequential composition, such as CCS [Mil80] and π -calculus [SW01].

Since its introduction, various papers have addressed the conservativeness of the well-branchedness condition in MPST. One line of research relaxes the condition by using a merge operation to allow all agents to have different behaviour in different branches, as long as they are informed of the choice in a timely manner [CYH09; Den+12; SY19]. Another line relaxes the condition by allowing different branches to start with different receivers, rather than enforcing the same receiver in every branch [DY12; CDP12; BMT14; CDG19; JY20]. These results may also serve as inspiration for relaxations of the well-branchedness condition on branching pomsets.

While our current conditions broadly correspond with well-formedness in MPST, we believe that our approach offers three advantages. First, as discussed before, it is syntax-oblivious, meaning it is not only applicable to MPST but to any specification which can be encoded as a branching pomset. Second, we believe that branching pomsets have the potential to be more expressive than global types in MPST. As mentioned above, our well-channeledness condition is already more liberal than the one in MPST. We have described various sources of possible relaxations for our well-branchedness condition, both in the MPST and in the pomset literature. Lastly, we conjecture that our tree-likeness condition is needed to simplify our proofs, and that it is possible — though more complex — to prove realisability without it (or at least with a relaxed version of it).

6.7 Conclusion

We have defined structural well-formedness conditions on BPs for communication protocols (Definition 6.12) and have proven that a well-formed BP

represents a realisable protocol ([Theorem 6.24](#)). The conditions are defined on the syntax of BPs, i.e., on the events, the labelling function, the causality relation and the branching structure, and their verification does not require exploring a BP's semantics; this is showcased in [Chapter 7](#). This answers [Question 3](#) positively. Furthermore, since these conditions are defined on BPs, they are generic: we can use them to reason about choreographies in the language defined in [Chapter 5](#) and any other language that can be incoded in BPs, without having to develop a specific analysis for the language in question. Our conditions are sufficient but not necessary, and relaxations of them are an important topic for future work. We have discussed several possible relaxations of well-branchedness in [Section 6.6](#). We now discuss the other three conditions.

Well-channeledness The BP R_h in [Figure 6.1](#) is not well-channeled since the events labelled with ab?int and ab?bool are unordered. It is unrealisable because a local system will force the int to be received before the bool while the global BP allows a different order. However, in this case one may take the view that the problem is not the local system being too strict, but rather the global BP being too liberal in an environment with ordered buffers: it should then in some way allow a user to specify just the two acceptable orderings without having to resort to adding a choice between the two and duplicating all events in the BP. Therefore, instead of changing the well-channeledness condition, another avenue could be to define alternative reduction rules for branching pomsets themselves ([Figure 4.2](#)), specifically tailored to ordered buffers. This could be done in such a way that reducing R_h by firing ab!int then automatically adds a dependency from ab?int to ab?bool . Alternatively, the semantics could be augmented with message queues to include the messages in transit. The latter would be the more straightforward option. This might allow the well-channeledness condition to be significantly relaxed or to possibly be removed altogether.

Tree-likeness Having the assumption of tree-likeness simplifies our proofs. It is our aim to eventually relax or even remove it and still prove realisability, but this will require a significantly more complex proof. We have noted in [Section 6.6](#) that global types in MPST and expressions in, e.g., CCS and the π -calculus, are tree-like by default. Conceptually a non-tree-like BP could potentially be turned into an equivalent (i.e., bisimilar) tree-like one by distributing the offending events over the branches of the involved choice. For example, consider the BP R_i in [Figure 6.1](#). By duplicating ab!int and ab?int and adding a copy of each with the relevant dependencies to each of the two branches of the

choice, we obtain a bisimilar but now tree-like (and well-formed) BP. A more general scheme may be developed based on versions of the CCS expansion theorem [HM85; FGM91]. However, regaining expressiveness at the cost of duplicating events effectively negates the benefits of using BPs in the first place.

Choreographicness The choreographicness condition works well for BPs derived from choreographies, but could eventually be relaxed to also allow more general BPs. Choreographicness forces the two events in a send-receive pair to be siblings in the branching structure. For example, in the BP R_i in Figure 6.1 it would be disallowed to distribute `ab!int` over the choice while leaving `ab?int` outside, even though the resulting BP would be bisimilar to R_i .

Implementation

This chapter is based on section 6 of our JLAMP 2024 paper [JLAMP24]. It also contains part of the conclusion of this paper. It strengthens the answer to Question 3 given in Chapter 6.

We developed a companion tool to simulate BPs and analyse their applications to choreographies. The tool is open-source²², developed in Scala and compiled into JavaScript (JS). A snapshot of a compiled JS can be executed from an internet browser at <https://lmf.di.uminho.pt/b-pomset/>. A subsequent paper [COORD23] presents the broader toolset of which the BP tool is an example. The BP tool and several other examples are included in the corresponding software artefact, which is available in the Zenodo repository at <https://doi.org/10.5281/zenodo.7888538>.

The tool fulfils three main objectives: (1) *internal validation*, providing us early insights over alternative notions of well-formedness, (2) *dissemination*, to better explain the proposed analysis using an interactive environment, and (3) *algorithmic insights* of well-formedness (cf. Section 6.3), by transforming the declarative definitions into concrete algorithms and identifying possible bottlenecks. This chapter focuses on the last two objectives, i.e., on *how to use* the online tool, and on *how to calculate* well-formedness.

²²<https://github.com/arcalab/choreo/tree/b-pomset>

Branching Pomsets Encoder

Choreography ↻

```
1 // Rh example
2 a->b:int; || a->b:bool
3 [1->3]
```

Examples ↑ ↓

Ra Rb Rc Rd Re Rf Rg Rh Ri
 Ri (tree-like) Review Review (choreographic)
 Review (strict) Buyer-seller Streaming
 BS-ill-chan SS-ill-chan MW DV Race
 C1 C2 ICE: Fig.5 ICE: Fig.6 ICE: Ex.4.1
 ICE: Ex.4.2 ICE: Ex.4.3 ATM Non-choreo

Rh example from the companion journal paper.
 Alice (a) sends a number followed by a boolean
 to Bob (b), and Bob receives these in any order.
 Not well-formed but realisable.

Sequence Diagram (Choreo only) ↓

```
sequenceDiagram
    participant a as a
    participant b as b
    par
        a->>b: int
        a->>b: bool
    end
```

Global B-Pomset ↓

```
graph TD
    1["1:ab!:int"] --> 2["2:ab?:int"]
    1 --> 3["3:ab!:bool"]
    3 --> 4["4:ab?:bool"]
```

B-Pomset Semantics ↻

Choreo Semantics (without added dependencies for b-pomsets) ↻

Well-formed

Well-branched
 NOT well-channelled:
 - Receiving events 2 and 4 are neither related nor disjoint.
 Tree-like
 Choreographic

Realisability via bisimulation

Local B-Pomset

Global LTS ↓

Global LTS info

Local LTS

Dependently Guarded (Choreo only)

Figure 7.1: Screenshot of the B-Pomset Encoder tool.

7.1 Using the B-Pomset Encoder

A screenshot of our analyser of BPs can be found in [Figure 7.1](#). The tool displays a collection of widgets, including the “Choreography” widget with an editor where the user describes the BP to be analysed. Each widget can be expanded or collapsed by clicking on the widget title; e.g., in the screenshot the “Choreography” widget is expanded and the “Global LTS” widget is collapsed. Clicking the refresh button at the top-right of the “Choreography” widget (or pressing Shift-Enter) updates all expanded widgets.

BPs are described using the choreography language in [Chapter 5](#), and are visualised in the “Global B-Pomset” widget after being encoded. Since some

BPs cannot be described by our choreography language, one can also write actions explicitly (e.g., “ $a!b:x$ ” represents the action $ab!x$ ²³) and can extend the causality relation (e.g., “[1->3]” denotes that the 1st event must precede the 3rd event), where 1 and 3 are the event identifiers depicted in the BP diagram. The BP R_i in Figure 6.1 can be written as both of the following:

“(a->b:yes + a->b:no) ; a->b:int”[☞]

“(a!b:yes|a?b:yes)+(a!b:no|a?b:no)||a!b:int||a?b:int[1->2,1->5,2->6,3->4,3->5,4->6,5->6]”[☞]

A list of different examples can be found in the “Examples” widget; clicking any of them will load this example to the “Choreography” widget.

The remaining widgets provide different analyses, including the following:

- “Global B-Pomset” draws the encoded BP from “Choreography”;
- “B-Pomset Semantics” allows a step-by-step exploration of the reduction semantics of an encoded BP (Figure 4.2), drawing the intermediate BPs;
- “Choreo Semantics (...)” allows a step-by-step exploration of the reduction semantics of the provided choreography (Figure 5.2);
- “Well-formed” checks if the BP is well-formed (Definition 6.12); it states for each well-formedness property whether it holds and gives a counter-example when it does not by pointing out violating events;
- “Realisability via bisimulation” searches for a bisimulation between the global LTS and the composition of the projected local LTSs (Definitions 6.1 and 6.4); this acts as our ground truth for realisability (Definition 6.2), but it is often infeasible due to state explosion;
- “Local B-Pomset” draws the projections of the BP on each process;
- “Global LTS” and “Local LTS” draw state machines using the semantics of BPs (Figure 4.2) applied to the global and to the projected BPs, respectively (bounding the number of states to a maximum value);
- “Sequence Diagram (...)” draws a sequence diagram representing the choreography (ignoring the extended causality arrows used to produce non-choreographic BPs).

²³We write $a!b:x$ instead of $ab!x$ to facilitate parsing.

Remark (loops). Our implementation targets only *finite* BPs. Hence, choreographies with loops are not covered by our analysis. However, the current version includes experiments with loops, i.e., some analyses will produce some results for BPs with special branches marked as loops. The theory for this extension is still under investigation and is not documented in this thesis.

7.2 Realising well-formedness

Our implementation of the algorithm that checks well-formedness realises the declarative definitions of each of the four sub-conditions (Definition 6.12). It provides some insights regarding the complexity of verifying these four sub-conditions. Recall that analysing realisability, as defined in Definition 6.2, requires traversing the full state-space of our semantics, which explodes exponentially in the presence of concurrent events. Our implementation avoids expanding the full state-space by combining multiple traversals over the BP graph structure, at the cost of rejecting certain realisable protocols. Below we sketch the implementation of each of these four sub-conditions, providing evidence that it is less complex than traversing all states.

- **Well-branched** (Definition 6.8) Our tool finds every choice with branches \mathcal{B}_1 and \mathcal{B}_2 . It proceeds by: (1) recursively checking if both branches are well-branched, (2) finding all *leaders* in each branch, i.e., events with no predecessor in the same branch, (3) collecting sending and receiving processes involved, and (4) verifying that only one leading sender and receiver are found (i.e., processes involved in events with no predecessors in the branch), that they are the same in both branches, and that they have different messages in the two branches. Furthermore, (5) each of these branches is projected to all remaining processes, and (6) corresponding projections from both branches are compared using graph isomorphism.²⁴

The causality relation is modelled as a hash-map from events to their predecessors (not being minimal nor transitively closed). Most operations are linear in the number of events, whereas the most complex operations are projecting both branches to all processes that are not in the leading events (5), and checking if each projection is isomorphic to its neighbour branch (6).

²⁴We used an existing algorithm for finding isomorphisms of acyclic graphs [Cor+96], applied to the predecessor relation.

- **Well-channeled** (Definition 6.9) Our tool starts by collecting, for each pair of processes \mathbf{a} and \mathbf{b} , the sets of channels $EM_{\mathbf{ab}}^!$ and $EM_{\mathbf{ab}}^?$, each consisting of events and messages (e, \mathbf{m}) sent by \mathbf{a} and received by \mathbf{b} , respectively. It then proceeds in two parts.

Firstly, for each distinct pair $(e_1, \mathbf{m}_1), (e_2, \mathbf{m}_2) \in EM_{\mathbf{ab}}^!$, it checks if e_1 and e_2 are either related ($e_1 \preceq e_2$ or $e_2 \preceq e_1$) or are mutually exclusive (in opposing branches of a choice). It also performs the same check for $EM_{\mathbf{ab}}^?$.

Secondly, for each set $EM_{\mathbf{ab}}^?$, it collects every event e paired with a direct predecessor e' , and for every distinct pair (e'_1, e_1) and (e'_2, e_2) from this set it checks if $e_1 \preceq e_2 \Rightarrow e'_1 \preceq e'_2$.

In these calculations, checking $e \preceq e'$ requires traversing the predecessors from e' until it finds e , and the *direct* predecessor relation is computed once by discarding redundant arcs in the predecessor relation. A worst case scenario is when the same channel appears in many places, leading to large sets $EM_{\mathbf{ab}}^!$ and $EM_{\mathbf{ab}}^?$. In turn, this would be a best case scenario to check well-branchedness, where fewer projections and graph isomorphism checks would be needed.

- **Tree-like** (Definition 6.10) Our tool starts by computing the successor relation by inverting the predecessor relation. It then checks, for every branch \mathcal{B} of every choice, if the set of all successors of its events includes elements outside that branch.
- **Choreographic** (Definition 6.11) Our tool pre-computes the successor relation (inverting the predecessor one), and for every event e proceeds in three parts.

Firstly, it traverses the predecessors until it finds, in every branch of this traversal, either (1) an event labelled with a sending action that matches the action of e , or (2) an event with the same subject.

Secondly, when e is labelled with some $\mathbf{ab}^?\mathbf{m}$ and is inside a branch of a choice, it checks if any of the neighbour events in the same branch (not further nested) is also a predecessor and labelled with a matching $\mathbf{ab}!\mathbf{m}$.

Thirdly, when e is labelled with some $\mathbf{ab}!\mathbf{m}$, it uses the successor relation to traverse all its successors. When finding an event with a matching $\mathbf{ab}^?\mathbf{m}$, our tool collects it, paired with the set of events $\mathbf{ab}!\mathbf{m}$ passed by during the traversal. It then checks if exactly one of these pairs has e as its only predecessor.

This informal explanation provides an overview of the set of traversals over the BP graph required to verify the four conditions that guarantee well-formedness. It also highlights the compactness of the formal definitions with respect to their realisation. Intuitively, we expect our implementation to have polynomial (time) complexity, corresponding to situations where, for every event, we need to reason over its neighbours in a branch or its predecessors (as in the choreographicness check). Furthermore, checking if a BP is well-channelled, when all its n events are labelled with sending actions $ab!m_i$ over the same agents and different messages m_i , would require comparing all pairs of messages (complexity $O(n^2)$). Each of these comparisons could further require traversing the predecessor relation, which in the worst case could mean traversing the n events. In any of these scenarios, the complexity is well below the exponential complexity required to analyse realisability via bisimulation²⁵, and can be validated by experimenting with the online tool.

7.3 Performance evaluation

We executed most examples mentioned in our JLAMP 2024 paper [JLAMP24], most of which also appear in this thesis, and measured the time to compute well-formedness and realisability (via bisimulation). We used the same source code, but compiled to and executed from a Java Virtual Machine, instead of using JavaScript. The results are presented in Table 7.1, using the abbreviations WB (well-branched), WC (well-channelled), TL (tree-like), and CH (choreographic). Each measurement was repeated ten times — we write $a \pm \delta$ to capture both the average a and the standard deviation δ . The ratio in the last column shows the average time required to check well-formedness relative to that required to check realisability via bisimulation; the lower the ratio, the larger the speed-up.

The experiments presented here do not aim at exploring corner cases regarding best- and worst-scenarios for well-formedness. Instead, the goal is to illustrate that, as discussed in Section 7.2, checking well-formedness does not suffer the same state explosion problem as a search for a bisimulation over the state-space.

Based on the results in Table 7.1, we draw three main conclusions.

1. The ratios in the rightmost column indicate that, in all examples, checking well-formedness is one order of magnitude faster than checking bisimulation-based realisability (ratio between 10%–100%), two orders

²⁵Note that checking bisimilarity can be done in polynomial time [BGS92]. However, computing the LTS may be exponential.

Table 7.1: Time (μs) to infer well-formedness and realisability via bisimulation of the examples in our JLAMP 2024 paper; values with **filled backgrounds** represent tests that reject the BP. The page numbers in the first column refer to the page number in this thesis, where applicable.

pg.	BP	Well-formed				Total	Realisable	Ratio
		WB	WC	TL	CH			
-	c_{fst}	282	204	39	177	703±133	53 833±13 795	1.30%
-	$c_{\text{strict}}^{\text{snd}}$	490	270	103	241	1 106±322	101 922±10 606	1.08%
-	$c_{\text{lenient}}^{\text{snd}}$	407	292	66	303	1 070±123	113 523±3 168	0.94%
92	c_1	79	81	34	84	279±98	5 759±1 372	4.84%
95	R_c	48	86	47	77	260±41	2 204±326	11.80%
95	R_d	53	109	43	153	359±42	7 121±341	5.05%
97	R_e	1	48	1	45	96±41	1 326±403	7.27%
104	R_f	63	94	34	72	265±65	2 253±164	11.77%
104	R_g	73	100	20	104	298±62	5 936±480	5.02%
104	R_h	1	38	3	35	78±46	702±91	11.10%
104	R_i	74	87	67	65	294±74	2 713±261	10.85%
108	R_i tree-like	101	110	30	90	333±74	2 846±445	11.69%
113	2-buyers	197	195	39	600	1 032±77	28 397±2 039	3.64%
113	streaming	1	140	1	220	364±190	56 151±2 532	0.65%
114	2-buyers ill	201	49	42	514	807±132	27 066±3 127	2.98%
114	streaming ill	1	96	1	100	200±65	8 371 116±201 610	0.00%

(1%–10%), or even three or more orders ($< 1\%$). Furthermore, we note that the speedup of checking well-formedness tends to be larger in cases where checking realisability is relatively expensive (i.e., relatively high values in the “Realisable” column). This suggests that checking well-formedness scales better than checking realisability.

2. In three examples, the check for well-formedness was negative, while the check for realisability was positive ($c_{\text{snd}}^{\text{strict}}$, $c_{\text{snd}}^{\text{lenient}}$, and R_i). This demonstrates that well-formedness is a sound, but incomplete, condition that conservatively approximates realisability. An important piece of future work is to make well-formedness more liberal while keeping favourable algorithmic properties.
3. Regarding the sub-conditions of well-formedness, checking tree-likeness tends to be the cheapest among the four. Neither one of the three other sub-conditions clearly stands out as being the most expensive one; it tends to depend on the structure of the BP under consideration.

We note that we consider these experimental results preliminary; a more extensive practical evaluation is needed to better understand, e.g., the performance of our approach in corner cases. However, we do believe that [Table 7.1](#) already gives an encouraging general impression.

7.4 Future work

We conclude this chapter with a brief discussion of future work.

Throughout this thesis we discuss a number of variations of BPs. In particular, BPs are currently unable to represent infinitely many sequences in a finite manner, as evidenced by the infinite unfolding in the encoding of loops in [Chapter 5](#). We discuss this further in [Chapter 8](#). Our current implementation includes some experimental support to represent loops symbolically, to enable analysis on them. We plan to further investigate the theory behind this, or the alternatives mentioned in [Chapter 8](#). Any other additions to the model, such as the n -ary choices mentioned in [Chapter 4](#), could also be added to the tool.

Conclusion

We started this thesis in [Chapter 1](#) by introducing the reader to choreographies. We presented their main benefits: designing distributed systems from a global perspective with a guarantee of certain safety properties. Conversely, we also discussed their drawbacks: the restricted expressiveness of existing choreography languages, and the realisability problem. We proceeded by posing three questions about these drawbacks, which we studied extensively in [Chapters 3](#) to [7](#). We conclude the thesis by summarising the answers to these questions, and we discuss several avenues for further research that we deem interesting.

8.1 Research questions

Question 1. Does there exist a choreography language capable of specifying all communication-error-free regular communication protocols?

[Chapter 3](#) answers this question positively, both for regular and ω -regular languages, the latter dealing with infinite words. Specifically, we leverage the shuffle on trajectories operator [[MRS98](#)] to answer an equivalent question in formal language theory. As discussed in [Chapter 3](#), this answer does not consider realisability, and the shuffle on trajectories operator, while versatile, is not straightforward to apply in practice for either design or analysis. Nevertheless, we believe that the theoretical results have merit, and that the shuffle on trajectories operator may be useful in the design of choreography languages by generalising other operators.

Question 2. Can we design a model that compactly represents combinations of concurrency and choice in choreographies?

[Chapter 4](#) introduces branching pomsets (BPs), which extend pomsets with a branching structure to encode choices. We also compare BPs with several classes of event structures (ESs), which extend posets with a different choice mechanism. BPs define an interesting new class of behaviour: while ESs use a more expressive choice mechanism, the enabling relation of BPs allows it to model complex dynamic behaviour. Consequently, BPs are incomparable with most existing classes of ESs, and are only contained in the most expressive classes of dynamic ESs. [Chapter 5](#) shows how BPs can be used to model the behaviour of choreographies, resulting in a compact model for choreographies containing both concurrency and choices.

Question 3. Can we define well-formedness conditions on branching pomsets such that they are both easily verifiable and sufficient to prove realisability?

[Chapter 6](#) defines well-formedness conditions on BPs and proves that any well-formed BP represents a realisable communication protocol. Since the conditions are structural, their verification can be performed by examining the events, their labels, the causality relation, and the branching structure; there is no exploration of the semantics. As further shown in [Chapter 7](#), this yields an efficient analysis. Furthermore, the conditions are defined on BPs and are thus generic: they can be applied to any choreography language that has a BP encoding, such as the one in [Chapter 5](#), without the need to develop any specific analysis for that language. The conditions are sufficient but not necessary, and there are (many) cases in which a realisable BP is not well-formed. We discuss possible relaxations of the conditions in [Chapter 6](#).

8.2 Future work

[Chapters 3](#) to [7](#) discuss various topics for further research specific to those chapters. Notably, we suggest further study of BPs and their relation with existing models of concurrency, and of relaxations of our well-formedness conditions. We conclude this chapter by discussing a number of further topics for future work.

Context-free choreographies While existing choreography languages are typically regular, recent years have also shown some interest in context-free languages, as evidenced by the work on context-free (binary) session types,

introduced by Thiemann and Vasconcelos [TV16; Pad17; Alm+21; SMV23]. It would be interesting to see if the shuffle on trajectories operator could contribute to the study of context-free choreography languages. As discussed in [Chapter 3](#), this is challenging since context-free languages are not closed under the shuffle on trajectories operator. A consideration could thus be to first study related language classes, in particular visibly pushdown languages [AM04], a subset of context-free languages that retains many of the desirable closure and decidability properties of regular languages.

Infinite BPs While BPs extend pomsets with choices, they do not add any mechanism for recursion or repetition. As such, encoding infinitely many possible (finite) sequences requires an infinitely large BP, as shown in the encoding of loops as infinitely unfolded choices in [Chapter 5](#). The same is true for ESs. All our proofs also hold for infinite BPs, but any analysis of these will have to be symbolic until BPs support a finite representation of infinite behaviour. We envision three possible directions for accomplishing this. One possibility would be to add an explicit repetition construct to the branching structure (e.g., change the second grammatical rule of [Definition 4.1](#) to $\mathcal{C} ::= e \mid \{\mathcal{B}_1, \mathcal{B}_2\} \mid \mathcal{B}^*$) and expand the semantics and proofs accordingly. Alternatively, a solution might be found in the extension from message sequence charts (MSCs) to MSC-graphs [ITU11]. A similar extension could be developed for BPs, where the BPs are sequentially composed in a (possibly cyclic) graph. Finally, it may be possible to leverage the more recently introduced pomset automata [Kap+19].

Well-formedness for event structures Because of the close relation between BPs and ESs, it is tempting to try to adapt the well-formedness conditions defined in [Chapter 6](#) to the latter. We believe, however, that this is not straightforward. Our well-branchedness condition in particular reasons over choices and their branches, which poses a problem in two ways: (1) ‘choices’ in the sense of well-branchedness are only represented implicitly in ESs, which would require extracting them first, and (2) as stated in [Chapter 4](#), ESs can describe more complex choices than BPs, which would require a different way of reasoning about them. A more promising approach might be to in some way apply the notion of ‘active’ and ‘passive’ agents in the well-formedness conditions for pomsets by Guanciale and Tuosto [TG18] to ESs. Well-formedness of ESs is also discussed by Castellani et al. in a recent paper on ES semantics for multiparty sessions [CDG23], where they also reference the conditions by Guanciale and Tuosto.

Encoding shuffles on trajectories as BPs The shuffle on trajectories operator could be used to increase the expressiveness of choreography languages. Simultaneously, models such as BPs and ESs can compactly model choreography behaviour. A major step forward would then be to join these two lines of research and find an encoding of shuffles on trajectories as, e.g., BPs. We note that a straightforward approach would be to, at least for regular languages, translate shuffles on trajectories to automata, the automata to regular expressions, and the regular expressions to BPs (cf. [Figure 5.4](#)). However, a BP obtained this way may grow very large, and the challenge is thus to find an efficient (and preferably direct) encoding. We recently presented a very first step in this direction, in which we explore the application of the shuffle on trajectories operator to individual posets [[iFM23](#)].

API generation In a recent paper [[ECOOP22](#)] we have investigated API generation for multiparty session types, and how two forms of combinatorial explosion can be avoided by leveraging both the pomset framework by Guanciale and Tuosto [[GT19](#)] and the match types feature in Scala 3. These APIs can verify at compile time whether an implementation is compliant with respect to a projected choreography. We believe that this can be further explored in the context of BPs.

Additional proofs for Chapter 3

Theorem 3.2. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton. Then M is balanced iff there exists a certificate $\nabla : Q \times \mathbb{N}^+ \rightarrow \mathbb{Z}$ such that, for all i :*

- $\nabla(q, i) \geq 0$ for all $q \in Q$;
- $\nabla(q, i) = 0$ for $q = q_0$ and for all $q \in F$; and
- if $(p, \sigma, q) \in \delta$ then $\nabla(q, i) = \nabla(p, i) + \nabla(\sigma, i)$, where $\nabla(\sigma, i) = 1$ if $\sigma = \lceil_i$, $\nabla(\sigma, i) = -1$ if $\sigma = \rfloor_i$ and $\nabla(\sigma, i) = 0$ otherwise.

Proof. We prove the two directions separately.

Certificate \rightarrow balanced. First we show that, if an automaton M satisfies the conditions stated in the theorem, then its language must be balanced. Note that, if M 's language is empty, then, by assumption, it consists of a single state and contains no transitions. It then clearly satisfies the conditions and its language is balanced. For the remainder of this proof, we assume that M 's language is not empty, that F is not empty, that every state in Q can be reached from q_0 and that every state in Q can reach some state in F .

We use the state elimination method by Brzozowski and McCluskey [BM63] to construct a regular expression e from M and then show that e is balanced. The construction extends the transitions of the automaton to contain not

only alphabet symbols but regular expressions. We thus extend the conditions stated in the theorem in two ways:

- We extend the third condition in the theorem to handle regular expressions: if $(p, e, q) \in \delta$ then $\nabla(q, i) = \nabla(p, i) + \nabla(e, i)$, with $\nabla(e, i)$ as defined in Figure 3.4. Note that this condition subsumes the earlier one, so M initially satisfies this new one as well.
- We add a fourth condition: if $(p, e, q) \in \delta$ then $\nabla^{\min}(e, i) + \nabla(p, i) \geq 0$. Again, M initially satisfies this condition: if $(p,]_i, q) \in \delta$ then, since $\nabla^{\min}(]_i, i) = -1$, $\nabla(q, i) = \nabla(p, i) - 1$ and $\nabla(q, i) \geq 0$, it follows that $\nabla^{\min}(]_i, i) + \nabla(p, i) \geq 0$. If $(p, e, q) \in \delta$ and $e \in \{\varepsilon,]_i,]_j\}$ with $j \neq i$, then $\nabla^{\min}(e, i) = 0$ and, since $\nabla(p, i) \geq 0$, it follows that $\nabla^{\min}(e, i) + \nabla(p, i) \geq 0$.

We show that these properties are retained by the state elimination method.

First, we amend M by adding a new initial state q'_0 with no incoming transitions and a single outgoing empty transition to the previous initial state q_0 . Similarly, we add a new state q_f with no outgoing transitions and an incoming empty transition from every final state in F ; we then change F so that q_f is the sole final state. Since the corresponding transitions are empty, we can assign $\nabla(q_0, i) = \nabla(q_f, i) = 0$ without violating any conditions. The automaton's language is unchanged.

Whenever we obtain two parallel transitions (p, e_1, q) and (p, e_2, q) , we replace them with a single transition $(p, e_1 + e_2, q)$. We then know that $\nabla(e_1, i) = \nabla(q, i) - \nabla(p, i) = \nabla(e_2, i)$ for all i , from which it follows that $\nabla(q, i) = \nabla(p, i) + \nabla(e_1 + e_2, i)$. Furthermore, as we have both $\nabla^{\min}(e_1, i) + \nabla(p, i) \geq 0$ and $\nabla^{\min}(e_2, i) + \nabla(p, i) \geq 0$, it also follows that $\nabla^{\min}(e_1 + e_2, i) + \nabla(p, i) = \min(\nabla^{\min}(e_1, i), \nabla^{\min}(e_2, i)) + \nabla(p, i) \geq 0$.

We pick any state q to be removed. Let e_2 be the expression on the self-loop for q if it has one. For any incoming transition (p, e_1, q) and outgoing transition (q, e_3, s) , we add the transition $(p, e_1 e_2^* e_3, s)$ (or $(p, e_1 e_3, s)$ if q has no self-loop), which replaces the path from p to s via q . We then remove q and all its transitions, and repeat this step until all states have been removed except for q_0 and q_f . The result is an automaton with a single transition (q_0, e, q_f) , where $L(e) = L(M)$.

If the removed state q has a self-loop, then

$$\begin{aligned}
 \nabla(s, i) &= \nabla(q, i) + \nabla(e_3, i) \\
 &= \nabla(p, i) + \nabla(e_1, i) + \nabla(e_3, i) \\
 &= \nabla(p, i) + \nabla(e_1, i) + \nabla(e_2^*, i) + \nabla(e_3, i) \\
 &\quad \text{(since } \nabla(e_2, i) = \nabla(e_2^*, i) = 0) \\
 &= \nabla(p, i) + \nabla(e_1 e_2^* e_3, i)
 \end{aligned}$$

Furthermore, either

- $\nabla^{\min}(e_1 e_2^* e_3, i) = \nabla^{\min}(e_1, i)$, in which case $\nabla^{\min}(e_1, i) + \nabla(p, i) \geq 0$;
or
- $\nabla^{\min}(e_1 e_2^* e_3, i) = \nabla(e_1, i) + \nabla^{\min}(e_2, i)$, in which case $\nabla(e_1, i) + \nabla^{\min}(e_2, i) + \nabla(p, i) = \nabla^{\min}(e_2, i) + \nabla(q, i) - \nabla(p, i) + \nabla(p, i) = \nabla^{\min}(e_2, i) + \nabla(q, i) \geq 0$; or
- $\nabla^{\min}(e_1 e_2^* e_3, i) = \nabla(e_1, i) + \nabla(e_2, i) + \nabla^{\min}(e_3, i)$. Since $\nabla(e_2, i) = 0$, analogously to the previous case $\nabla(e_1, i) + \nabla^{\min}(e_3, i) + \nabla(e_2, i) = \nabla^{\min}(e_3, i) + \nabla(q, i) \geq 0$.

If q has no self-loop then, by the above, $\nabla(s, i) = \nabla(p, i) + \nabla(e_1 e_3, i)$ and $\nabla^{\min}(e_1 e_3, i) + \nabla(p, i) \geq 0$.

Finally, for the resulting expression e , we then have $\nabla(e, i) = \nabla(q_f, i) - \nabla(q_0, i) = 0 - 0 = 0$ and $\nabla^{\min}(e, i) = \nabla^{\min}(e, i) + \nabla(e, i) \geq \nabla(q_0, i) = 0$. It follows that $\nabla^{\min}(e, i) = 0$, and then, by [Theorem 3.7](#), e is balanced.

Balanced \rightarrow certificate. We now show that, if M is balanced, then there exists a certificate satisfying the stated conditions. If M 's language is empty, then, by assumption, it consists of a single state q and contains no transitions. Assigning $\nabla(q, i) = 0$ then satisfies all conditions. For the remainder of this proof, we assume that M 's language is not empty, that F is not empty, that every state in Q can be reached from q_0 and that every state in Q can reach some state in F .

Let $i \in \mathbb{N}^+$. We iteratively assign an i -balance to the states in Q , starting with $\nabla(q_0, i) = 0$. Then, as long as there exists a transition (p, σ, q) , where p has an i -balance and q does not, we assign $\nabla(q, i) = \nabla(p, i) + \nabla(\sigma, i)$, with $\nabla(\sigma, i)$ as in [Figure 3.4](#). Since every state in Q is reachable from q_0 , this process will eventually assign an i -balance to every state, which will match

the i -balance of some word leading to it. Note that every word leading to a state of M is a prefix of some word in $L(M)$.

Suppose, for the sake of contradiction, that the constructed certificate violates one of the conditions stated in the theorem:

- If $\nabla(q, i) < 0$ for some $q \in Q$, then some word in $L(M)$ has a prefix with a negative i -balance, and M is unbalanced.
- If $\nabla(q, i) \neq 0$ for some $q \in F$ (recall that we fixed $\nabla(q_0, i) = 0$, so we can omit that possible violation), then M accepts an unbalanced word and M is unbalanced.
- If $\nabla(q, i) \neq \nabla(p, i) + \nabla(\sigma, i)$ for some $(p, \sigma, q) \in \delta$, then q can be reached through words with different i -balances, and then M will accept words through words with different i -balances. In other words: M is unbalanced.

Any violation of the conditions on ∇ will thus also violate our assumption that M is balanced. Since this holds for any i , we conclude that the stated conditions hold for every i . \square

Theorem 3.15. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a Muller automaton. M is balanced iff there exist lower and upper bounds on the i -balance of every state in Q (respectively $\nabla^L(q, i)$ and $\nabla^U(q, i)$ for a state q) such that, for every i ,*

- (i) $\nabla^L(q_0, i) = \nabla^U(q_0, i) = 0$;
- (ii) $\nabla^L(q, i) \geq 0$ for every $q \in Q$;
- (iii) $\nabla^L(q, i) \leq \nabla^L(p, i) + \nabla(\sigma, i) \leq \nabla^U(p, i) + \nabla(\sigma, i) \leq \nabla^U(q, i)$ for every $(p, \sigma, q) \in \delta$; and
- (iv) for every $\{f_1, \dots, f_\ell\} \in F$ (representing a nonempty language), either there exist some j, k such that $(\nabla^L(f_j, i), \nabla^U(f_j, i)) \neq (\nabla^L(f_k, i), \nabla^U(f_k, i))$, or $\nabla^L(f_j, i) = \nabla^U(f_j, i) = 0$ for every j .

Proof. We prove the two directions separately.

Bounds \rightarrow balanced. First we prove that, if there exist bounds as described, then M is balanced. The general idea is to assume that $|F| = 1$

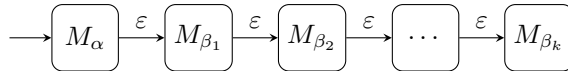
and apply the construction by McNaughton [McN66] to obtain an automaton accepting roughly the same language, to which we apply M 's certificate. Using the state elimination method by Brzozowski and McCluskey [BM63], as featured in the proof of Theorem 3.2, we transform this automaton into two regular expressions e_α, e_β , such that $e_\alpha e_\beta^\omega = L(M)$. The corresponding balance follows from the construction and the original certificate of M .

If M 's language is empty, then it is balanced by default. If M 's language is not empty, then it is the union of the languages of the Muller automata $(Q, \Sigma, \delta, q_0, \{f\})$ for all $f \in F$. We can thus assume, without loss of generality, that $|F| = 1$.

Let $\{f_1, \dots, f_k\}$ be the single element in F . As per McNaughton's construction, let α be the language of all the words taking M from q_0 to state f_1 and, for every $1 \leq i \leq k$, let β_i be the language of all words taking M from state f_c to state f_{i+1} — or from f_k to f_1 if $i = k$ — without visiting any states other than those in $\{f_1, \dots, f_k\}$. It follows that $L(M) = \alpha(\beta_1\beta_2 \dots \beta_k)^\omega$.

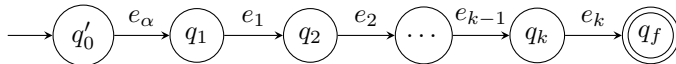
By definition, α is accepted by the finite automaton $M_\alpha = (Q, \Sigma, \delta, q_0, \{f_1\})$. For every i , β_i is accepted by the finite automaton $M_{\beta_i} = (\{f_1, \dots, f_k\}, \Sigma, \delta', \{f_i\}, \{f_{i+1}\})$, the last member being f_1 for M_k . Here δ' consists exactly of all transitions in δ between states in $\{f_1, \dots, f_k\}$.

We construct a finite automaton M' by concatenating M_α, M_{β_i} : we add an empty transition from the final state of M_α to the initial state of M_{β_1} , and one from the final state of M_{β_i} to the initial state of $M_{\beta_{i+1}}$ for every $i < k$. Then $L(M') = \alpha\beta_1\beta_2 \dots \beta_k$.

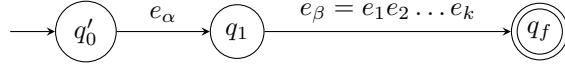


Note that, since all states and transitions in M' are copies of those in M , and the states linked by the new empty transitions are copies of the same state in M , we can assign to every state the balance tuple of the corresponding original state in M , which then satisfies conditions (i, ii, iii).

By applying the state elimination method, we can reduce M' to an automaton M'' with $k + 2$ states and $k + 1$ transitions:



In this automaton M'' , q'_0 and q_f are the new initial respectively final state added by the construction. Since q'_0 is added with an empty transition to the initial state of M' , q_0 , and no incoming transitions, it is consistent to assign $\nabla(q'_0, i) = \nabla(q_0, i) = (0, 0)$. Similarly, since q_f is added with an empty transition from the final state of M' , f_1 , and no outgoing transitions, it is consistent to assign $\nabla(q_f, i) = \nabla(f_1, i)$. For every $1 \leq i \leq k$, q_i is the initial state of M_{β_i} . The automaton is obtained by eliminating all other states. It can then be reduced even further to an automaton M''' with three states and two transitions, by eliminating states q_2, \dots, q_k :



By this construction, $L(e_\alpha) = \alpha$ and $L(e_i) = \beta_i$ for every $1 \leq i \leq k$, and thus $L(e_\alpha e_\beta^\omega) = \alpha(\beta_1 \beta_2 \dots \beta_k)^\omega = L(M)$.

A straightforward inductive argument (analogous to the one in the proof of [Theorem 3.2](#)) will show that the following properties are retained by the state elimination method:

For every $(p, e, q) \in \delta$ and for every i ,

- (a) $\nabla^L(q, i) \leq \nabla^L(p, i) + \nabla^L(e, i)$;
- (b) $\nabla^U(q, i) \geq \nabla^U(p, i) + \nabla^U(e, i)$; and
- (c) $\nabla^{\min}(e, i) + \nabla^L(p, i) \geq 0$.

Let $n \in \mathbb{N}^+$. As stated before, $\nabla(q'_0, n) = (0, 0)$. Since q_1 is the initial state of M_{β_1} , i.e., f_1 , and $\nabla(q_f, n) = \nabla(f_1, n)$, it follows that $\nabla(e_\beta, n) = (0, 0)$. By condition (iv), we can distinguish two cases:

- $\nabla(f_i, n) = (0, 0)$ for every $1 \leq i \leq k$, in which case it follows from $\nabla(q'_0, n) = (0, 0)$ and (iii) that $\nabla(e_\alpha, n) = (0, 0)$, and then $\nabla(e_\alpha e_\beta^\omega, n) = (0, 0)$.
- $\nabla(f_i, n) \neq \nabla(f_j, n)$ for some $i \neq j$. Without loss of generality, let $i < j$. A straightforward argument by contradiction gives us that $\nabla(e_i \dots e_{j-1}, n) = (\ell, \ell)$ for some $\ell \neq 0$. It then follows from [Lemma 3.17\(ii\)](#) and [Lemma 3.16\(i\)](#) that $\xi(e_i \dots e_{j-1}, n)$, and then that $\xi(e_\beta, n)$, that $\xi^\omega(e_\beta^\omega, n)$ and therefore that $\nabla(e_\alpha e_\beta^\omega, n) = (0, 0)$.

In both cases, $\nabla(e_\alpha e_\beta^\omega, n) = (0, 0)$. Since $\nabla^{\min}(e_\alpha, n) \geq 0 - \nabla^L(q_0, n) = 0$ and

$$\begin{aligned} \nabla^{\min}(e_\beta, n) + \nabla^L(e_\alpha, n) &\geq 0 - \nabla^L(f_1, n) + \nabla^L(e_\alpha, n) && \text{(c)} \\ &\geq 0 - \nabla^L(q_0, n) - \nabla^L(e_\alpha, n) + \nabla^L(e_\alpha, n) && \text{(a)} \\ &\geq 0 && (\nabla^L(q_0, n) = 0) \end{aligned}$$

and $\nabla^{\min}(e, n) \leq 0$ for all e , it follows that $\nabla^{\min}(e_\alpha e_\beta^\omega, n) = 0$. Since this holds for all n , it follows from [Theorem 3.19](#) that $e_\alpha e_\beta^\omega$ is balanced and then so is M .

Balanced \rightarrow bounds. We now prove that, if M is balanced, then there exist lower and upper bounds on the balances of the states of M as described in the theorem. For each state $q \in Q$, we can obtain a regular expression e_q describing all the finite words leading to it — for example using the Brzozowski and McCluskey state elimination method. To each state we assign the balance bounds of the corresponding regular expression, as defined in [Figure 3.18](#). We show that this assignment satisfies the conditions stated in the theorem.

First note that these regular expressions describe sets of prefixes of words in $L(M)$; it then follows that they must satisfy the conditions of [Lemma 3.18](#), which then states that their bounds are defined. For the remainder of this proof, fix i . We now turn to the four conditions:

- (ii) Let $q \in Q$. Recall that e_q describes a set of prefixes of words in $L(M)$, from which it follows that the i -balance of any word in $L(e_q)$ is at least 0. Then, by [Lemma 3.17](#)(iii), $\nabla^L(e_q, i) \geq 0$ and, consequently, $\nabla^L(q, i) \geq 0$.
- (i) Since $\varepsilon \in L(e_{q_0})$, it follows from [Lemma 3.17](#)(ii) that $\nabla^L(q_0, i) \leq 0$. Recall from (ii) that $\nabla^L(q_0, i) \geq 0$. Then $\nabla^L(q_0, i) = 0$.

Suppose, for the sake of contradiction, that $\nabla^U(q_0, i) > 0$. It follows from [Lemma 3.17](#)(iii) that there is a path from q_0 to q_0 with more opening than closing i -parentheses. Since there is no limit on the number of times this path may be travelled, this would mean that $L(M)$ is unbounded and thus, contrary to our premise, also unbalanced.

- (iii) Let $(p, \sigma, q) \in \delta$. It follows from [Lemma 3.17\(iii\)](#) that there exists some $w \in L(e_p)$ such that $\nabla(w, i) = \nabla^L(e_p, i) = \nabla^L(p, i)$. Then $w\sigma \in L(e_q)$ and, consequently, it follows from [Lemma 3.17\(ii\)](#) that $\nabla^L(e_q, i) \leq \nabla(w\sigma, i) = \nabla^L(p, i) + \nabla(\sigma, i)$. Analogously, $\nabla^U(p, i) + \nabla(\sigma, i) \leq \nabla^U(q, i)$. Finally, it follows from [Lemma 3.17\(ii\)](#) that $\nabla^L(e_p, i) \leq \nabla^U(e_p, i)$ and thus that $\nabla^L(p, i) + \nabla(\sigma, i) \leq \nabla^U(p, i) + \nabla(\sigma, i)$.
- (iv) Let $\{f_1, \dots, f_\ell\} \in F$ represent a nonempty language. Suppose, for the sake of contradiction, that $\nabla(f_1, i) = \dots = \nabla(f_\ell, i) \neq (0, 0)$, i.e., neither of the cases specified in the theorem hold. We can then focus on $\nabla(f_1, i)$. Since $\nabla^U(f_1, i) \geq \nabla^L(f_1, i)$, it follows that $\nabla^U(f_1, i) > 0$. It follows from [Lemma 3.17\(iii\)](#) that there exists some word leading to f_1 with a nonzero i -balance. Since f_2, \dots, f_ℓ have the same i -balances as f_1 , there cannot be any transitions labelled with i -parentheses within $\{f_1, \dots, f_\ell\}$. Consequently, the acceptance criterion $\{f_1, \dots, f_\ell\}$ will lead M to accept a word with a finite, nonzero number of unmatched opening i -parentheses, which contradicts our premise that M is balanced.

Since this holds for any i , we conclude that the assigned balance bounds satisfy all conditions stated in the theorem. \square

Dynamic causality event structures

In this appendix we compare the expressiveness of dynamic causality event structures (DCEs) [Arb+18] with that of the variant with counters (DCCESs) defined in Definition 4.21, and with branching pomsets (BPs) as defined in Definitions 4.27 and 4.28. Specifically, we show that DCEs are incomparable with both.

We first give the original semantics of DCEs. Recall that DCEs and DCCESs share the same syntax and only differ in semantics. Recall from Definitions 4.20 and 4.21 and the discussion preceding them that, for DCCESs, an event e causes another event e' at a given stage of computation if the causality has been added more times than that it has been dropped. In other words: the causality relation at a given stage of computation is independent of the order in which the preceding events occurred. In contrast, causalities in DCEs are order-sensitive: the last addition or drop determines whether a causality is present at the current stage. Therefore, configurations in DCEs need to be paired with an ordering.

In the following definition, a state in the transition system is a pair (X, \rightarrow_X) , where \rightarrow_X represents the current causality relation. \triangleleft_X and \blacktriangleright_X represent the sets of dependencies dropped respectively added by the events in X . Condition 2 ensures both consistency and securing. It also implies that two events occurring in the same step cannot be causally related, so they can only be concurrent. Condition 3 specifies the update of the current causality relation after a

transition. Conditions 4 and 5 only apply to steps containing more than one event: the first requires that only dependencies that are ineffective (because one of the two related events has already occurred) can be simultaneously added and dropped in the same step, while the latter requires that if a dependency $[e \rightarrow e_2]$ is added in the same step where the caused event e_2 occurs, then the cause e should have occurred previously in the computation.

Definition B.1 (DCES semantics [Arb+18]). Let $S = (E, \rightarrow, \triangleleft, \blacktriangleright)$ be a dynamic causality event structure and let $X, Y \subseteq E$. Letting $\rightarrow_\emptyset = \rightarrow$, we define $(X, \rightarrow_X) \models_S (Y, \rightarrow_Y)$ if

1. $X \subset Y$;
2. $\forall e \in Y \setminus X : \{e' \mid (e', e) \in \rightarrow_X\} \subseteq X$;
3. $\rightarrow_Y = (\rightarrow_X \setminus \triangleleft_{Y \setminus X}) \cup \blacktriangleright_{Y \setminus X}$;
4. $\forall (e_1, e_2) \in \blacktriangleright_{Y \setminus X} \cap \triangleleft_{Y \setminus X} : (e_1 \in X \vee e_2 \in X)$; and
5. $\forall e_1, e_2 \in Y \setminus X : \forall e \in E : e_1 \blacktriangleright [e \rightarrow e_2] \implies e \in X$.

We first show that DCESs are incomparable with DCCESs.

Lemma B.2. *Dynamic causality event structures are incomparable with dynamic causality event structures with counters.*

Proof. Let $S = (E, \rightarrow, \triangleleft, \blacktriangleright)$ where $E = \{e_1, e_2, e_3, e, e'\}$, $\rightarrow = \emptyset$ and $e_1 \blacktriangleright [e \rightarrow e']$, $e_2 \triangleleft [e \rightarrow e']$, $e_3 \blacktriangleright [e \rightarrow e']$.

First, interpret S as a DCES and assume, for the sake of contradiction, that there exists some DCCES $S' = (E, \rightarrow', \triangleleft', \blacktriangleright')$ such that $S \simeq_t S'$. Note that this implies trace equivalence of S and S' . Since e_1, e_2, e_3, e and e' are all traces of S and therefore of S' , it follows that $\rightarrow' = \emptyset$. Furthermore, e_1e_2 and e_2e_1 are both traces of S and therefore of S' , so neither adds a dependency to the other. Then, since e_1e' is no trace of S , e_1 must add some dependency to e' . However, e_1e_2e' is a trace of S and therefore of S' , meaning e_2 must drop whatever dependency was added to e' by e_1 , or e_2 itself is the dependency in question. It follows from all of the preceding that e_2e_1e' must also be a trace of S' , while it is not a trace of S .

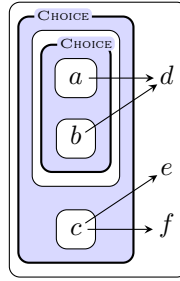
Next, interpret S as a DCCES and assume, for the sake of contradiction, that

there exists some DCES $S' = (E, \rightarrow', \triangleleft', \blacktriangleright')$ such that $S \simeq_t S'$. Again, this implies trace equivalence of S and S' . As before, because all the singleton events are traces of S , it follows that $\rightarrow' = \emptyset$. Since $e_1 e'$ is no trace of S , e_1 must add some dependency to e' . Since $e e_1 e'$ is a trace of S , this dependency must be e . However, since $e_2 e_1 e'$ is also a trace of S , the dependency must also be e_2 . Since e and e_2 are two distinct events, this is contradictory. \square

We now show that DCESs are incomparable with BPs.

Lemma B.3. *Dynamic causality event structures are incomparable with branching pomsets.*

Proof. Since PESs are not included in BPs ([Lemma 4.29](#)) and PESs and included in DCESs, it follows trivially that DCESs are not included in BPs. For the other direction, consider the BP R below:



For the sake of contradiction, assume that there exists a DCES $S = (E, \rightarrow, \triangleleft, \blacktriangleright)$ such that $S \simeq_t R$. Note that this also implies trace equivalence of S and R . Since a, b, c, d, e and f are all traces of R (and thus of S), it follows that $\rightarrow = \emptyset$. Since ed is not a trace of R , it then follows that e adds some dependency to d . Since $faed$ is a trace of R , it follows that this dependency cannot be b, c or d ; similarly, since $fbed$ is a trace of R , it cannot be a . By process of elimination, we conclude that $e \blacktriangleright [f \rightarrow d]$ and that e adds no other dependency to d . Analogously, $f \blacktriangleright [e \rightarrow d]$ and f adds no other dependency to d . It then follows that $\{e, f\} \mapsto_S \{e, f, d\}$, but $\{e, f\} \not\mapsto_R \{e, f, d\}$. Since $\{e, f\}$ is reachable in \mapsto_R , and thus also in \mapsto_S , this is contradictory. \square

We note that the BP R in [Lemma B.3](#) can be represented as a DCCES, although not trivially; it is thus not a counterexample to [Conjecture 4.36](#). The early reasoning from [Lemma B.3](#) still holds: in any equivalent DCCES, there are

still no initial dependencies, and both e and f must add some dependency to d . Since efd is a trace of R , e and f must add some dependency to d other than each other. Since we are now considering a DCCES, and since both aed and afd are traces of R , it follows that a must either drop any dependency added by e and f , or be the dependency itself. However, a can only drop any dependency once, and since $aefd$ is a trace of R , it follows that e and f cannot add the same dependencies to d — both adding a as a dependency is prevented by the analogous case for b .

The “trick” to the solution is to add these dependencies indirectly: have e and f add each other as a dependency to d , and have one of the two also add a (and optionally b). In turn, a and b drop all (non-self) dependencies once, which cancels out since no dependency is added more than once. Formally, this corresponds to the following: $e \blacktriangleright [a, b, f \rightarrow d]$, $f \blacktriangleright [e \rightarrow d]$, $a \blacktriangleleft [b, e, f \rightarrow d]$, $b \blacktriangleleft [a, e, f \rightarrow d]$, where $e \blacktriangleright [a, b, f \rightarrow d]$ is a shorthand for the addition of the three dependencies from a , b and f to d , and analogously for the others. This corresponds to the behaviour of R .

References

(Citing pages are listed after each entry.)

- [AEY03] Rajeev Alur, Kousha Etessami and Mihalis Yannakakis. ‘Inference of Message Sequence Charts’. In: *IEEE Trans. Software Eng.* 29.7 (2003), pp. 623–633. DOI: [10.1109/TSE.2003.1214326](https://doi.org/10.1109/TSE.2003.1214326) (Page 114).
- [AEY05] Rajeev Alur, Kousha Etessami and Mihalis Yannakakis. ‘Realizability and verification of MSC graphs’. In: *Theor. Comput. Sci.* 331.1 (2005), pp. 97–114. DOI: [10.1016/j.tcs.2004.09.034](https://doi.org/10.1016/j.tcs.2004.09.034) (Pages 7, 16).
- [Alm+21] Bernardo Almeida, Andreia Mordido, Peter Thiemann and Vasco T. Vasconcelos. ‘Polymorphic Context-free Session Types’. In: *CoRR* abs/2106.06658 (2021). arXiv: [2106.06658](https://arxiv.org/abs/2106.06658) (Page 129).
- [AM04] Rajeev Alur and P. Madhusudan. ‘Visibly pushdown languages’. In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. Ed. by László Babai. ACM, 2004, pp. 202–211. DOI: [10.1145/1007352.1007390](https://doi.org/10.1145/1007352.1007390) (Pages 59, 129).
- [Arb+18] Youssef Arbach, David S. Karcher, Kirstin Peters and Uwe Nestmann. ‘Dynamic Causality in Event Structures’. In: *Log. Methods Comput. Sci.* 14.1 (2018). DOI: [10.23638/LMCS-14\(1:17\)2018](https://doi.org/10.23638/LMCS-14(1:17)2018) (Pages 68, 69, 72, 73, 76, 77, 79, 139, 140).
- [Bar+23] Adam D. Barwell, Ping Hou, Nobuko Yoshida and Fangyi Zhou. ‘Designing Asynchronous Multiparty Protocols with Crash-Stop Failures’. In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*. Ed. by Karim Ali and Guido Salvaneschi.

- Vol. 263. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 1:1–1:30. DOI: [10.4230/LIPICSECOOP.2023.1](https://doi.org/10.4230/LIPICSECOOP.2023.1) (Page 13).
- [BBO12] Samik Basu, Tefvik Bultan and Meriem Ouederni. ‘Deciding choreography realizability’. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 191–202. DOI: [10.1145/2103656.2103680](https://doi.org/10.1145/2103656.2103680) (Page 114).
- [BC87] Gérard Boudol and Ilaria Castellani. ‘On the semantics of concurrency: partial orders and transition systems’. In: *TAPSOFT*. Ed. by Hartmut Ehrig, Robert A. Kowalski, Giorgio Levi and Ugo Montanari. Vol. 249. LNCS. Heidelberg: Springer, 1987, pp. 123–137 (Page 72).
- [BC88] Gérard Boudol and Ilaria Castellani. ‘Permutation of transitions: an event structure semantics for CCS and SCCS’. In: *REX: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. Ed. by Jaco W. de Bakker, Willem P. de Roever and Grzegorz Rozenberg. Vol. 354. LNCS. Heidelberg: Springer, 1988, pp. 411–427 (Page 70).
- [BC91] Gérard Boudol and Ilaria Castellani. *Flow models of distributed computations: event structures and nets*. Research Report 1482. INRIA, 1991 (Pages 70, 74).
- [BCM01] Paolo Baldan, Andrea Corradini and Ugo Montanari. ‘Contextual Petri Nets, Asymmetric Event Structures, and Processes’. In: *Information and Computation* 171.1 (2001), pp. 1–49. DOI: <https://doi.org/10.1006/inco.2001.3060> (Pages 71, 75).
- [BGS92] José L. Balcázar, Joaquim Gabarró and Miklos Santha. ‘Deciding Bisimilarity is P-Complete’. In: *Formal Aspects Comput.* 4.6A (1992), pp. 638–648 (Page 124).
- [BM63] Janusz A. Brzozowski and Edward J. McCluskey. ‘Signal Flow Graph Techniques for Sequential Circuit State Diagrams’. In: *IEEE Trans. Electron. Comput.* 12.2 (1963), pp. 67–76 (Pages 131, 135).
- [BMT14] Laura Bocchi, Hernán C. Melgratti and Emilio Tuosto. ‘Resolving Non-determinism in Choreographies’. In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014,*

- Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 493–512. DOI: [10.1007/978-3-642-54833-8_26](https://doi.org/10.1007/978-3-642-54833-8_26) (Page 116).
- [BZ83] Daniel Brand and Pitro Zafropulo. ‘On Communicating Finite-State Machines’. In: *J. ACM* 30.2 (1983), pp. 323–342. DOI: [10.1145/322374.322380](https://doi.org/10.1145/322374.322380) (Page 102).
- [CDG19] Iliaria Castellani, Mariangiola Dezani-Ciancaglini and Paola Giannini. ‘Reversible sessions with flexible choices’. In: *Acta Informatica* 56.7-8 (2019), pp. 553–583. DOI: [10.1007/s00236-019-00332-y](https://doi.org/10.1007/s00236-019-00332-y) (Page 116).
- [CDG23] Iliaria Castellani, Mariangiola Dezani-Ciancaglini and Paola Giannini. ‘Event structure semantics for multiparty sessions’. In: *J. Log. Algebraic Methods Program.* 131 (2023), p. 100844. DOI: [10.1016/J.JLAMP.2022.100844](https://doi.org/10.1016/J.JLAMP.2022.100844) (Page 129).
- [CDP12] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini and Luca Padovani. ‘On Global Types and Multi-Party Session’. In: *Log. Methods Comput. Sci.* 8.1 (2012). DOI: [10.2168/LMCS-8\(1:24\)2012](https://doi.org/10.2168/LMCS-8(1:24)2012) (Page 116).
- [CM13] Marco Carbone and Fabrizio Montesi. ‘Deadlock-freedom-by-design: multiparty asynchronous global programming’. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 263–274. DOI: [10.1145/2429069.2429101](https://doi.org/10.1145/2429069.2429101) (Pages 7, 90).
- [CM20] Luís Cruz-Filipe and Fabrizio Montesi. ‘A core model for choreographic programming’. In: *Theor. Comput. Sci.* 802 (2020), pp. 38–66. DOI: [10.1016/j.tcs.2019.07.005](https://doi.org/10.1016/j.tcs.2019.07.005) (Page 7).
- [Cor+96] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone and Mario Vento. ‘An efficient algorithm for the inexact matching of ARG graphs using a contextual transformational model’. In: *13th International Conference on Pattern Recognition, ICPR 1996, Vienna, Austria, 25-19 August, 1996*. IEEE Computer Society, 1996, pp. 180–184. DOI: [10.1109/ICPR.1996.546934](https://doi.org/10.1109/ICPR.1996.546934) (Page 122).
- [CS59] N. Chomsky and M.P. Schützenberger. ‘The Algebraic Theory of Context-Free Languages’. In: *Computer Programming and Formal Systems*. Vol. 26. Studies in Logic and the Foundations of Mathematics. Elsevier, 1959, pp. 118–161 (Page 28).

- [CYH09] Marco Carbone, Nobuko Yoshida and Kohei Honda. ‘Asynchronous Session Types: Exceptions and Multiparty Interactions’. In: *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures*. Ed. by Marco Bernardo, Luca Padovani and Gianluigi Zavattaro. Vol. 5569. Lecture Notes in Computer Science. Springer, 2009, pp. 187–212. DOI: [10.1007/978-3-642-01918-0_5](https://doi.org/10.1007/978-3-642-01918-0_5) (Page 116).
- [CZ97] Ilaria Castellani and Guo-Qiang Zhang. ‘Parallel Product of Event Structures’. In: *Theor. Comput. Sci.* 179.1-2 (1997), pp. 203–215. DOI: [10.1016/S0304-3975\(96\)00104-1](https://doi.org/10.1016/S0304-3975(96)00104-1) (Page 88).
- [Den+12] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri and Raymond Hu. ‘Parameterised Multiparty Session Types’. In: *Log. Methods Comput. Sci.* 8.4 (2012). DOI: [10.2168/LMCS-8\(4:6\)2012](https://doi.org/10.2168/LMCS-8(4:6)2012) (Page 116).
- [Dom04] Michael Domaratzki. ‘Trajectory-Based Embedding Relations’. In: *Fundam. Informaticae* 59.4 (2004), pp. 349–363 (Page 39).
- [Duc00] Philippe Duchon. ‘On the enumeration and generation of generalized Dyck words’. In: *Discret. Math.* 225.1-3 (2000), pp. 121–135. DOI: [10.1016/S0012-365X\(00\)00150-3](https://doi.org/10.1016/S0012-365X(00)00150-3) (Page 28).
- [DY12] Pierre-Malo Deniélou and Nobuko Yoshida. ‘Multiparty Session Types Meet Communicating Automata’. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 194–213. DOI: [10.1007/978-3-642-28869-2_10](https://doi.org/10.1007/978-3-642-28869-2_10) (Page 116).
- [FBS04] Xiang Fu, Tevfik Bultan and Jianwen Su. ‘Conversation protocols: a formalism for specification and verification of reactive electronic services’. In: *Theor. Comput. Sci.* 328.1-2 (2004), pp. 19–37. DOI: [10.1016/j.tcs.2004.07.004](https://doi.org/10.1016/j.tcs.2004.07.004) (Page 114).
- [FGM91] Gian Luigi Ferrari, Roberto Gorrieri and Ugo Montanari. ‘An Extended Expansion Theorem’. In: *TAPSOFT’91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD)*.

- Ed. by Samson Abramsky and T. S. E. Maibaum. Vol. 494. Lecture Notes in Computer Science. Springer, 1991, pp. 29–48. DOI: [10.1007/3540539816_56](https://doi.org/10.1007/3540539816_56) (Page 118).
- [FJ23] Francisco Ferreira and Sung-Shik Jongmans. ‘Oven: Safe and Live Communication Protocols in Scala, using Synthetic Behavioural Type Analysis’. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17–21, 2023*. Ed. by René Just and Gordon Fraser. ACM, 2023, pp. 1511–1514. DOI: [10.1145/3597926.3604926](https://doi.org/10.1145/3597926.3604926) (Page 13).
- [FL17] Alain Finkel and Étienne Lozes. ‘Synchronizability of Communicating Finite State Machines is not Decidable’. In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10–14, 2017, Warsaw, Poland*. Ed. by Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn and Anca Muscholl. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 122:1–122:14. DOI: [10.4230/LIPIcs.ICALP.2017.122](https://doi.org/10.4230/LIPIcs.ICALP.2017.122) (Page 114).
- [Gis88] Jay L. Gischer. ‘The Equational Theory of Pomsets’. In: *Theor. Comput. Sci.* 61 (1988), pp. 199–224. DOI: [10.1016/0304-3975\(88\)90124-7](https://doi.org/10.1016/0304-3975(88)90124-7) (Page 81).
- [GP04] Rob J. van Glabbeek and Gordon D. Plotkin. ‘Event Structures for Resolvable Conflict’. In: *Mathematical Foundations of Computer Science 2004, 29th International Symposium, MFCS 2004, Prague, Czech Republic, August 22–27, 2004, Proceedings*. Ed. by Jirí Fiala, Václav Koubek and Jan Kratochvíl. Vol. 3153. Lecture Notes in Computer Science. Springer, 2004, pp. 550–561. DOI: [10.1007/978-3-540-28629-5_42](https://doi.org/10.1007/978-3-540-28629-5_42) (Pages 74, 78, 79).
- [Gru71] Jozef Gruska. ‘A Characterization of Context-free Languages’. In: *J. Comput. Syst. Sci.* 5.4 (1971), pp. 353–364 (Page 59).
- [GT19] Roberto Guanciale and Emilio Tuosto. ‘Realisability of pomsets’. In: *J. Log. Algebraic Methods Program.* 108 (2019), pp. 69–89. DOI: [10.1016/j.jlamp.2019.06.003](https://doi.org/10.1016/j.jlamp.2019.06.003) (Pages 13, 14, 103, 107, 114, 115, 130).
- [HM85] Matthew Hennessy and Robin Milner. ‘Algebraic Laws for Nondeterminism and Concurrency’. In: *J. ACM* 32.1 (1985), pp. 137–161. DOI: [10.1145/2455.2460](https://doi.org/10.1145/2455.2460) (Page 118).
- [HY16] Raymond Hu and Nobuko Yoshida. ‘Hybrid Session Verification Through Endpoint API Generation’. In: *Fundamental*

- Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings.* Ed. by Perdita Stevens and Andrzej Wasowski. Vol. 9633. Lecture Notes in Computer Science. Springer, 2016, pp. 401–418. DOI: [10.1007/978-3-662-49665-7_24](https://doi.org/10.1007/978-3-662-49665-7_24) (Page 13).
- [HYC08] Kohei Honda, Nobuko Yoshida and Marco Carbone. ‘Multiparty asynchronous session types’. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008.* Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 273–284. DOI: [10.1145/1328438.1328472](https://doi.org/10.1145/1328438.1328472) (Pages 7, 17, 90, 100, 101, 106, 113, 115).
- [HYC16] Kohei Honda, Nobuko Yoshida and Marco Carbone. ‘Multiparty Asynchronous Session Types’. In: *J. ACM* 63.1 (2016), 9:1–9:67. DOI: [10.1145/2827695](https://doi.org/10.1145/2827695) (Page 7).
- [ITU11] ITU. *ITU-T recommendation Z.120. Message Sequence Chart (MSC)*. 2011 (Pages 7, 129).
- [JF23] Sung-Shik Jongmans and Francisco Ferreira. ‘Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types’. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 42:1–42:30. DOI: [10.4230/LIPIcs.ECOOP.2023.42](https://doi.org/10.4230/LIPIcs.ECOOP.2023.42) (Page 116).
- [JY20] Sung-Shik Jongmans and Nobuko Yoshida. ‘Exploring Type-Level Bisimilarity towards More Expressive Multiparty Session Types’. In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 251–279. DOI: [10.1007/978-3-030-44914-8_10](https://doi.org/10.1007/978-3-030-44914-8_10) (Pages 13, 116).
- [Kap+19] Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva and Fabio Zanasi. ‘On series-parallel pomset languages: Rationality, context-freeness and automata’. In: *J. Log. Algebraic Methods*

- Program.* 103 (2019), pp. 130–153. DOI: [10.1016/j.jlamp.2018.12.001](https://doi.org/10.1016/j.jlamp.2018.12.001) (Page 129).
- [KL98] Joost-Pieter Katoen and Lennard Lambert. ‘Pomsets for MSC’. In: *Formale Beschreibungstechniken für verteilte Systeme, 8. GI/ITG-Fachgespräch, Cottbus, 4. und 5. Juni 1998*. Ed. by Hartmut König and Peter Langendörfer. Verlag Shaker, 1998, pp. 197–207 (Page 90).
- [Kle56] Stephen C. Kleene. ‘Representation of events in nerve nets and finite automata’. In: *Automata studies* (1956) (Page 11).
- [Kou+18] Dimitrios Kouzapas, Ornela Dardha, Roly Perera and Simon J. Gay. ‘Typechecking protocols with Mungo and StMungo: A session type toolchain for Java’. In: *Sci. Comput. Program.* 155 (2018), pp. 52–75. DOI: [10.1016/J.SCICO.2017.10.006](https://doi.org/10.1016/J.SCICO.2017.10.006) (Page 13).
- [Lan92] Rom Langerak. ‘Transformations and Semantics for LOTOS’. In: 1992 (Pages 71, 74–76).
- [Lan93] Rom Langerak. ‘Bundle Event Structures: a Non-Interleaving Semantics for LOTOS’. In: *Formal Description Techniques for Distributed Systems and Communication Protocols*. Ed. by Michael Diaz and Roland Groz. Amsterdam: North-Holland, 1993, pp. 331–346 (Page 70).
- [Lie03] Jens Liebehenschel. ‘Lexicographical Generation of a Generalized Dyck Language’. In: *SIAM J. Comput.* 32.4 (2003), pp. 880–903. DOI: [10.1137/S0097539701394493](https://doi.org/10.1137/S0097539701394493) (Page 28).
- [Loh02] Markus Lohrey. ‘Safe Realizability of High-Level Message Sequence Charts’. In: *CONCUR 2002 - Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings*. Ed. by Lubos Brim, Petr Jancar, Mojmir Kretínský and Antonín Kucera. Vol. 2421. Lecture Notes in Computer Science. Springer, 2002, pp. 177–192. DOI: [10.1007/3-540-45694-5_13](https://doi.org/10.1007/3-540-45694-5_13) (Page 16).
- [LW09] Niels Lohmann and Karsten Wolf. ‘Realizability Is Controllability’. In: *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers*. Ed. by Cosimo Laneve and Jianwen Su. Vol. 6194. Lecture Notes in Computer Science. Springer, 2009, pp. 110–127. DOI: [10.1007/978-3-642-14458-5_7](https://doi.org/10.1007/978-3-642-14458-5_7) (Page 114).

- [LY90] Jacques Labelle and Yeong-Nan Yeh. ‘Generalized Dyck paths’. In: *Discret. Math.* 82.1 (1990), pp. 1–6. DOI: [10.1016/0012-365X\(90\)90039-K](https://doi.org/10.1016/0012-365X(90)90039-K) (Page 28).
- [Mar+99] Carlos Martín-Vide, Alexandru Mateescu, Grzegorz Rozenberg and Arto Salomaa. ‘Contexts on trajectories’. In: *Int. J. Comput. Math.* 73.1 (1999), pp. 15–36 (Page 59).
- [McN66] Robert McNaughton. ‘Testing and generating infinite sequences by a finite automaton’. In: *Information and control* 9.5 (1966), pp. 521–530 (Pages 24, 52, 135).
- [Mil80] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. DOI: [10.1007/3-540-10235-3](https://doi.org/10.1007/3-540-10235-3) (Pages 24, 116).
- [Mon23] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023 (Pages 10, 114).
- [Moo14] Michael Moortgat. ‘A Note on Multidimensional Dyck Languages’. In: *Categories and Types in Logic, Language, and Physics*. Vol. 8222. Lecture Notes in Computer Science. Springer, 2014, pp. 279–296. DOI: [10.1007/978-3-642-54789-8_16](https://doi.org/10.1007/978-3-642-54789-8_16) (Page 28).
- [MRS98] Alexandru Mateescu, Grzegorz Rozenberg and Arto Salomaa. ‘Shuffle on Trajectories: Syntactic Constraints’. In: *Theor. Comput. Sci.* 197.1-2 (1998), pp. 1–56. DOI: [10.1016/S0304-3975\(97\)00163-1](https://doi.org/10.1016/S0304-3975(97)00163-1) (Pages 13, 35, 37, 39, 50, 51, 59, 127).
- [MSY00] Alexandru Mateescu, Kai Salomaa and Sheng Yu. ‘On Fairness of Many-Dimensional Trajectories’. In: *J. Autom. Lang. Comb.* 5.2 (2000), pp. 145–157. DOI: [10.25596/jalc-2000-145](https://doi.org/10.25596/jalc-2000-145) (Page 35).
- [Mul63] David E. Muller. ‘Infinite sequences and finite machines’. In: *SWCT*. IEEE Computer Society, 1963, pp. 3–16 (Page 23).
- [NPW81] Mogens Nielsen, Gordon Plotkin and Glynn Winskel. ‘Petri Nets, Event Structures and Domains, Part I’. In: *Theoretical Computer Science* 13.1 (1981), pp. 85–108 (Pages 16, 68).
- [OS05] Alexander Okhotin and Kai Salomaa. ‘Contextual Grammars with Uniform Sets of Trajectories’. In: *Fundam. Informaticae* 64.1-4 (2005), pp. 341–351 (Page 59).
- [Pad17] Luca Padovani. ‘Context-Free Session Type Inference’. In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by

- Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 804–830. DOI: [10.1007/978-3-662-54434-1_30](https://doi.org/10.1007/978-3-662-54434-1_30) (Page 129).
- [Pra86] Vaughan R. Pratt. ‘Modeling concurrency with partial orders’. In: *Int. J. Parallel Program.* 15.1 (1986), pp. 33–71. DOI: [10.1007/BF01379149](https://doi.org/10.1007/BF01379149) (Pages 14, 25).
- [Pro65] Pierre Probst. *Caroline à la mer*. Éditions Hachette, 1965 (Page 1).
- [Pro79] Helmut Prodinger. ‘On a generalization of the Dyck-language over a two letter alphabet’. In: *Discret. Math.* 28.3 (1979), pp. 269–276. DOI: [10.1016/0012-365X\(79\)90134-1](https://doi.org/10.1016/0012-365X(79)90134-1) (Page 28).
- [RW01] Arend Rensink and Heike Wehrheim. ‘Process algebra with action dependencies’. In: *Acta Informatica* 38.3 (2001), pp. 155–234. DOI: [10.1007/s002360100070](https://doi.org/10.1007/s002360100070) (Pages 89, 90, 93, 100).
- [SAB20] Klaus-Dieter Schewe, Yamine Aït Ameur and Sarah Benyagoub. ‘Realisability of Choreographies’. In: *Foundations of Information and Knowledge Systems - 11th International Symposium, FoIKS 2020, Dortmund, Germany, February 17-21, 2020, Proceedings*. Ed. by Andreas Herzig and Juha Kontinen. Vol. 12012. Lecture Notes in Computer Science. Springer, 2020, pp. 263–280. DOI: [10.1007/978-3-030-39951-1_16](https://doi.org/10.1007/978-3-030-39951-1_16) (Page 114).
- [SMV23] Gil Silva, Andreia Mordido and Vasco T. Vasconcelos. ‘Subtyping Context-Free Session Types’. In: *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium*. Ed. by Guillermo A. Pérez and Jean-François Raskin. Vol. 279. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 11:1–11:19. DOI: [10.4230/LIPICS.CONCUR.2023.11](https://doi.org/10.4230/LIPICS.CONCUR.2023.11) (Page 129).
- [SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001 (Page 116).
- [SY19] Alceste Scalas and Nobuko Yoshida. ‘Less is more: multiparty session types revisited’. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 30:1–30:29. DOI: [10.1145/3290343](https://doi.org/10.1145/3290343) (Pages 13, 116).
- [TG18] Emilio Tuosto and Roberto Guanciale. ‘Semantics of global view of choreographies’. In: *J. Log. Algebraic Methods Program.* 95 (2018), pp. 17–40. DOI: [10.1016/j.jlamp.2017.11.002](https://doi.org/10.1016/j.jlamp.2017.11.002) (Pages 115, 129).
- [TV16] Peter Thiemann and Vasco T. Vasconcelos. ‘Context-free session types’. In: *Proceedings of the 21st ACM SIGPLAN Inter-*

- national Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller and Eijiro Sumii. ACM, 2016, pp. 462–475. DOI: [10.1145/2951913.2951926](https://doi.org/10.1145/2951913.2951926) (Page 129).
- [Win80] Glynn Winskel. ‘Events in Computation’. PhD thesis. University of Edinburgh, 1980 (Pages 68–70, 74, 75).
- [Win82] Glynn Winskel. ‘Event structure semantics for CCS and related languages’. In: *ICALP*. Ed. by Mogens Nielsen and Erik Meineche Schmidt. Vol. 140. LNCS. Heidelberg: Springer, 1982, pp. 561–576 (Page 70).
- [WN93] Glynn Winskel and Mogens Nielsen. ‘Models for concurrency’. In: (1993) (Page 88).
- [YZF21] Nobuko Yoshida, Fangyi Zhou and Francisco Ferreira. ‘Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types’. In: *Fundamentals of Computation Theory - 23rd International Symposium, FCT 2021, Athens, Greece, September 12-15, 2021, Proceedings*. Ed. by Evripidis Bampis and Aris Pagourtzis. Vol. 12867. Lecture Notes in Computer Science. Springer, 2021, pp. 18–35. DOI: [10.1007/978-3-030-86593-1_2](https://doi.org/10.1007/978-3-030-86593-1_2) (Page 13).

Author publications

(Citing pages are listed after each entry.)

- [COORD23] José Proença and Luc Edixhoven. ‘Caos: A Reusable Scala Web Animator of Operational Semantics’. In: *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*. Ed. by Sung-Shik Jongmans and Antónia Lopes. Vol. 13908. Lecture Notes in Computer Science. Springer, 2023, pp. 163–171. DOI: [10.1007/978-3-031-35361-1_9](https://doi.org/10.1007/978-3-031-35361-1_9) (Pages [20](#), [119](#)).
- [DLT21] Luc Edixhoven and Sung-Shik Jongmans. ‘Balanced-By-Construction Regular and ω -Regular Languages’. In: *Developments in Language Theory - 25th International Conference, DLT 2021, Porto, Portugal, August 16-20, 2021, Proceedings*. Ed. by Nelma Moreira and Rogério Reis. Vol. 12811. Lecture Notes in Computer Science. Springer, 2021, pp. 130–142. DOI: [10.1007/978-3-030-81508-0_11](https://doi.org/10.1007/978-3-030-81508-0_11) (Pages [18](#), [19](#), [27](#)).
- [DLT21 TR] Luc Edixhoven and Sung-Shik Jongmans. *Balanced-by-construction regular and ω -regular languages (technical report)*. Tech. rep. OUNL-CS-2021-1. Open University of the Netherlands, 2021 (Page [27](#)).
- [ECOOP22] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans and José Proença. ‘API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3’. In: *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*. Ed. by Karim Ali and Jan Vitek. Vol. 222. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 27:1–27:28. DOI: [10.4230/LIPICS.ECOOP.2022.27](https://doi.org/10.4230/LIPICS.ECOOP.2022.27) (Pages [19](#), [20](#), [130](#)).
- [FACS22] Luc Edixhoven and Sung-Shik Jongmans. ‘Realisability of Branching Pomsets’. In: *Formal Aspects of Component Software - 18th International Conference, FACS 2022, Virtual Event, November 10-11, 2022, Proceedings*. Ed. by Silvia Lizeth Tapia Tarifa and José Proença. Vol. 13712. Lecture Notes in Com-

- puter Science. Springer, 2022, pp. 185–204. DOI: [10.1007/978-3-031-20872-0_11](https://doi.org/10.1007/978-3-031-20872-0_11) (Pages 19, 101).
- [FACS22 TR] Luc Edixhoven and Sung-Shik Jongmans. *Realisability of branching pomsets (technical report)*. Tech. rep. OUNL-CS-2022-05. Open University of the Netherlands, 2022 (Page 101).
- [ICE22] Luc Edixhoven, Sung-Shik Jongmans, José Proença and Guillermina Cledou. ‘Branching Pomsets for Choreographies’. In: *Proceedings 15th Interaction and Concurrency Experience, ICE 2022, Lucca, Italy, 17th June 2022*. Ed. by Clément Aubert, Cinzia Di Giusto, Larisa Safina and Alceste Scalas. Vol. 365. EPTCS. 2022, pp. 37–52. DOI: [10.4204/EPTCS.365.3](https://doi.org/10.4204/EPTCS.365.3) (Pages 19, 61, 89).
- [ICE22 TR] Luc Edixhoven, Sung-Shik Jongmans, José Proença and Guillermina Cledou. *Branching pomsets for choreographies (technical report)*. Tech. rep. OUNL-CS-2022-04. Open University of the Netherlands, 2022 (Page 89).
- [iFM23] Luc Edixhoven. ‘Shuffling Posets on Trajectories’. In: *iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands, November 13-15, 2023, Proceedings*. Ed. by Paula Herber and Anton Wijs. Vol. 14300. Lecture Notes in Computer Science. Springer, 2023, pp. 384–390. DOI: [10.1007/978-3-031-47705-8_21](https://doi.org/10.1007/978-3-031-47705-8_21) (Pages 20, 130).
- [IJFCS23] Luc Edixhoven and Sung-Shik Jongmans. ‘Balanced-by-Construction Regular and ω -Regular Languages’. In: *Int. J. Found. Comput. Sci.* 34.2&3 (2023), pp. 117–144. DOI: [10.1142/S0129054122440026](https://doi.org/10.1142/S0129054122440026) (Pages 19, 27).
- [JLAMP24] Luc Edixhoven, Sung-Shik Jongmans, José Proença and Ilaria Castellani. ‘Branching pomsets: Design, expressiveness and applications to choreographies’. In: *J. Log. Algebraic Methods Program.* 136 (2024), p. 100919. DOI: [10.1016/J.JLAMP.2023.100919](https://doi.org/10.1016/J.JLAMP.2023.100919) (Pages 19, 61, 89, 101, 119, 124).

Summary

Distributed computing is becoming ever more important. However, designing and implementing distributed systems correctly is notoriously difficult. An important factor in this is the interplay between participants in such a system: certain tasks are allowed concurrently, while others are dependent on each other. Coordinating participants requires communication between them, which can be specified using communication protocols. Choreographies are a specific notation for communication protocols, which specifies them from a global perspective and have the benefit of guaranteeing certain desirable properties by construction. As a drawback, there are also more restrictions on the communication patterns they can specify. The first part of this thesis contributes to removing these restrictions. We study an existing operator from formal language theory, called shuffle on trajectories, and show how it can be used in a choreography language to specify a large class of behaviour, while preserving desirable properties.

A second problem is that it may be impossible to implement certain communication patterns in a way that precisely matches a specification; we then say that that specification is not realisable. In general, determining realisability takes exponential time or is undecidable. The second part of this thesis contributes to the development of a rich yet polynomial realisability analysis. We introduce a new model for the behaviour of choreographies, based on partially ordered sets: these ‘branching pomsets’ are capable of compactly modelling combinations of concurrency and choice, which typically occur in choreographies. We then compare this model with the existing literature on event structures, a class of related models. We define well-formedness conditions on the structure of branching pomsets, which are easy to check, and we show that they are sufficient to prove realisability of the modelled communication protocol. Finally, we discuss an implementation of branching pomsets and the described analysis.

Samenvatting

Gedistribueerde systemen worden almaar belangrijker. Het correct ontwerpen en implementeren ervan is echter erg ingewikkeld. Een belangrijke reden hiervoor is het samenspel van de deelnemers in een systeem: bepaalde taken mogen gelijktijdig gebeuren, terwijl andere taken afhankelijk zijn van elkaar. Het coördineren van deelnemers vereist communicatie tussen hen, wat kan worden gespecificeerd door middel van communicatieprotocollen. Choreografieën zijn een specifieke notatie voor communicatieprotocollen, die ze specificeert vanuit een globaal perspectief, en die bepaalde wenselijke eigenschappen per constructie garandeert. Als nadeel zijn er ook meer beperkingen op de communicatiepatronen die ze kunnen specificeren. Het eerste deel van dit proefschrift draagt bij aan het wegnemen van deze beperkingen. We bestuderen een bestaande operator uit de formeletalentheorie, met de naam ‘shuffle on trajectories’, en laten zien hoe deze kan worden gebruikt in een choreografietaal om een grote gedragsklasse te specificeren, met behoud van wenselijke eigenschappen.

Een tweede probleem is dat het onmogelijk kan zijn om bepaalde communicatiepatronen te implementeren op een manier die precies overeenkomt met een specificatie; we zeggen dan dat een specificatie niet realiseerbaar is. In het algemeen kost het bepalen van realiseerbaarheid exponentieel veel tijd of is het onbeslisbaar. Het tweede deel van dit proefschrift draagt bij aan de ontwikkeling van een uitgebreide maar polynomiale realiseerbaarheidsanalyse. We introduceren een nieuw model voor het gedrag van choreografieën, gebaseerd op partieel geordende verzamelingen: deze ‘vertakkende pomsets’ vormen een compact model voor combinaties van gelijktijdigheid en keuze, een typisch element van choreografieën. We vergelijken dit model met de bestaande literatuur over ‘event structures’, een klasse van gerelateerde modellen. We definiëren welgevormdheidseisen op de structuur van vertakkende pomsets, die eenvoudig

zijn om na te gaan, en we tonen aan dat deze afdoende zijn om te bewijzen dat het gemodelleerde communicatieprotocol realiseerbaar is. Ten slotte bespreken we een implementatie van vertakkende pomsets en de beschreven analyse.

Acknowledgements

First and foremost, I thank my supervisor, Sung. He has consistently supported me during these past five years, in particular in research and academic matters, but also in matters related to my career. A lot has happened since I started my PhD, often good, more than once not, and I am glad and grateful that he was there with me, be it in person or remotely, every step of the way. I have greatly enjoyed our time together, both professionally and personally, and I will always think fondly of the days he and I spent brainstorming in front of a white board — or just chatting about something completely unrelated while enjoying coffee and tea, respectively!

I thank my promoters, Bastiaan and Marcello. Although both of them joined as promoters late into my PhD, I am grateful for their guidance in the final year. Bastiaan, as chair of the computer science group, showed great understanding for my personal situation in late 2021 and early 2022. If Marcello had not urged me to take a closer look at Sung’s PhD position back in 2019, I likely would not have started in the first place.

I thank my co-authors, José, Guille and Ilaria. I always looked forward to our frequent online meetings; I am happy to have also met them in person, José and Ilaria several times. It was a pleasure working with them, and I hope to be able to do so again once a good opportunity arises!

I thank Hendrik Jan, Benjamin and Hans-Dieter for their contributions to the work presented in [Chapter 3](#).

I thank the members of the assessment committee for their time and comments. It cannot have been easy to read 150 pages of frequently dense material.

I thank my paranymphs, Tom and Coen, for standing by me — also literally!

I thank the members of the FM/CS and SWAT groups at CWI, where I physically carried out most of my research. I fondly recall the many interactions and laughs, in particular during the lunches and the typical discussions afterwards, about research, academia, society, and whatever else came up that day. I also thank the supporting staff I interacted with, all of whom contributed to making CWI such a pleasant working environment; in particular our secretaries, those manning the reception, library and cafeteria, and Minnie, whose job I cannot describe but whose sunny smile would always brighten my day!

I thank the members of the computer science department at the Open University, where I was employed during my PhD. Although I typically did not see them on a daily (or weekly) basis, I enjoyed our interactions at the regular department meetings (sometimes followed by dinner), at the online research seminar, and later every couple of weeks at the study centre in Utrecht. I also thank the supporting staff I interacted with, in particular our secretaries, Kees, those manning the study centre in Utrecht, and lately those in the beadle's office.

I thank my colleagues at LIACS, where I physically wrote my thesis during the last year of my PhD, while also working there as a lecturer. These two occupations would often interleave, and those I interacted with were typically interested in both, making it difficult to classify interactions as PhD-related or not. I am grateful to the management team for granting me the full-time use of a desk, considering office space was scarce in our building. I thank in particular the members of the theory group, as well as many other PhD students, postdocs, lecturers and professors, for their warm welcome and the many pleasant interactions. I also thank the supporting staff I interacted with, in particular our secretaries (whose office also provided tea, fruit and a laugh) and those manning the reception.

I thank the management team and members of the research school IPA, and the organisers and attendees of the NetTCS seminar, both of which provided me with opportunities to learn and to interact with peers within the Netherlands. I thank the organisers and participants of the summer school PLISS (2022), which, on top of learning and interaction, also provided a great holiday in Italy.

I thank, in general, the various colleagues in my research area whom I have interacted with these past five years, both within the Netherlands and abroad, for the warm welcome they have given me into their midst. When contrasted with the stories I sometimes hear about academia, I consider myself very lucky to have found such a friendly and supportive community.

Aside from the academic community, I also thank my family and friends — I consider myself very lucky (again) to have many of both. I cannot overstate their importance in my life, especially these past five years. Many positive things have happened since I started my PhD, but these years also saw the demise of several close family members, and a pandemic which forced me to work from home for over a year. I am extremely grateful to all those who supported me during those times, either directly or indirectly (and knowingly or not), often by simply distracting me with something fun or kind, and to all those who made the good times even better by spending their time with me, in person and online. I fondly recall family gatherings, both large and small, games, films, bowling, restaurants, holidays, summer camps, museums, many more things, and above all just lazing about, doing nothing in particular, and enjoying each other's company. I hope to keep doing so, far, far into the future!

Many have also shown genuine interest in my academic career and in my research, the latter of which is not easy without a background in mathematics or computer science. I hope that, over the years, I have gotten better at explaining my work in 'normal' words, although it remains a worthy challenge.

I specifically thank my parents, Bas and Reinie, my brother, Tom, and our cats, Zazie and Zen; my close family in Den Haag, Heiloo, Utrecht and Bussum; my extended family, spanning much of the south and west of the Netherlands; my friends from Het Duivelsei, in particular the (extended) 'Machiavelli' group; my study friends; my friends from Vierkant voor Wiskunde; my friends from high school; our many family friends, mostly mathematicians; our kind neighbours; and anyone close to me not covered by the aforementioned groups.

Finally, I thank my local tea (and coffee) shop, Het Klaverblad, whose tea blends are a daily blessing both at the university and at home; my barber, Manfred, who consistently delivers high quality; several local bakeries, pastry shops, ice cream shops, chocolatiers, and combinations thereof, whose delicious products I regularly enjoy; my dentist, Anouk; several local restaurants, where I always leave satisfied; and my local comic book shop, Dumpie, where I would happily spend the remainder of my salary, if only I had no need for savings and unlimited space for bookcases!



Curriculum Vitae

Luc Edixhoven was born in Rennes, France, on 23 July 1995. He (was) moved to Leiden, the Netherlands, in 2002, where, in 2013, he graduated from the Stedelijk Gymnasium Leiden. He then studied computer science at LIACS, Leiden University, where he obtained his bachelor and master degrees cum laude, respectively in 2016 and 2019. During his studies, Luc was an active member of the student board game association Het Duivelsei, where he served on the board and in several committees; he also volunteered as counsellor at the summer camps of Vierkant voor Wiskunde, which he still does.

In the fall of 2019, Luc joined the Open University of the Netherlands as a PhD candidate, under the supervision of Sung-Shik Jongmans. He first occupied an office at CWI, in Amsterdam, and later at LIACS, in Leiden. In the summer of 2022, he started working part time as a lecturer at LIACS, stretching the remainder of his PhD at the Open University into a longer, part-time appointment, until the summer of 2024. Currently, Luc is wrapping up his teaching duties in Leiden and is preparing to continue his academic career. In the spring of 2025, he will spend some months in Odense, Denmark. His plans for the more distant future, decidedly optimistic, remain to be determined.

Titles in the IPA Dissertation Series since 2021

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical.* Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition.* Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems.* Faculty

of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction.* Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software.* Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting.* Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement.* Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies.* Faculty of Science, OU. 2024-11

