Independence, name-passing and constraints in models for concurrency.

Håkon Normann

September 18, 2017

ABSTRACT

In this thesis we look at independence, name-passing and constraints in models for concurrency, with the goal of obtaining models that encompass all three aspects.

With independence we mean that two actions are independent if they have no impact on each other. Name-passing is that names or terms are communicated around in the process by actions, and can give rise to new actions or synchronizations. Constraints are when some conditions have to be met by the state of the process in order for some action to be enabled or disabled. We choose Psi-calculus as a common frame as this is a meta-model that encompass name-passing, constraints and parallel composition, but lack the notion of independence.

For the initial look on constraints we provide instantiations of event structures and DCR-graphs into Psi-calculi to prove that the assertions and constraints of Psi can encompass known constraint-based models with independence. We provide a notion of syntactically correct processes for each of the instantiations and a mapping from the event structure or DCR-graph into a syntactically correct process. We also show that any syntactically correct process is a mapping of an event structure or DCR-graph. For the event structure instantiation we give a notion of refinement of the process, which takes a syntactically correct process and generate a new syntactically correct process in the same style of standard event structure action refinement.

We then go on to provide non-interleaving semantics for the guarded early pi-calculus. Our starting point here is the non-interleaving semantics for CCS by Mukund and Nielsen, where the so-called structural (prefixing or subject) causality and events are defined from a notion of locations derived from the syntactic structure of the process term. The semantics are conservatively extended with a notion of extruder histories, from which we infer the so-called link (name or object) causality and events introduced by the name-passing topology of the pi-calculus. We prove that these semantics give rise to a labelled asynchronous transition system.

We finally introduce the logical causality that is generated by the assertions, conditions and enabling mechanics of the Psi-calculus, and what the obstacles that have to be overcome in order to get non-interleaving semantics for the full Psi-calculi framework.

ABSTRACT

I denne afhandling betragter vi uafhængighed, kommunikation af navne og logiske betingelser i modeller for samtidige systemer, med målet at kunne give modeller der tager højde for alle tre aspekter.

Med uafhængighed mener vi at to handlinger er uafhængige, hvis de ikke har nogle indflydelse på hinanden. Kommunikation af navne betyder at navne eller termer med navne kan blive kommunikeret mellem processer som handlinger, hvilket kan give anledning til at nye processer kan kommunikerer og synkronisere deres handlinger. Med logiske betingelser menes, at muligheden for at udføre en handling kan være betinget af at tilstanden af systemet opfylder en logisk betingelse. Vi vælger Psi-kalkuli som en fælles ramme for dette studie, da denne er en meta-model som tager højde for både kommunikation af navne, logiske betingelser og samtidige handlinger, men mangler en repræsentation af uafhængighed.

Som vores første studie af logiske betingelser giver vi vi instanser af event-strukturer og DCR-grafer i Psi-kaluli for at bevise at Psi-kalkulis mulighed for at benytte betingelser kan udtrykke kendte modeller med både betingelser og uafhængighed.

Vi går videre og giver en uafhængigheds-semantik for en variant af pi-kalkylen med såkaldt tidlig-semantik. Vores udgangspunkt er uafhængigheds-semantikken for CCS givet af Mukund og Nielsen, hvor såkaldt strukturel kausalitet og hændelser er defineret fra et lokationsbegreb udledt fra den syntaktiske struktur af proces-termerne. Endelig introducerer vi den logiske kausilitet som bliver genereret af de logiske betingelser i Psi-kalkuli og diskuterer hvilke udfordringer der udestår for at kunne give en uafhængigheds-semantik for Psi-kalkuli.

Acknowledgements

My Thanks

To my supervisor Thomas for his guidance in research, and for helping me when things got though both with work and life in general.

To my family for all the support, being there when I needed someone to talk to and always having a place i could stay when going to Oslo.

To the members of my quiz team Zero Knowledge for getting me out of the apartment.

To the students and staff at ITU for making it a great place to work, and study.

A final thanks to the Copenhagen psychiatric service for the much needed help the last year.

CONTENTS

1. Introduction . . . 8
1.1 Introduction . . . 8
1.2 Semantic models of non-interleaving . . . 9
1.2.1 Interleaving vs non-interleaving semantics . . . 9
1.2.2 Transition systems and Asynchronous transition systems . . . 9
1.2.3 Event Structures . . . 11
1.2.4 Action refinement . . . 14
1.2.5 DCR-graphs . . . 15
1.3 Process models . . . 18
1.3.1 Pi-calculus . . . 18
1.3.2 Psi-calculus . . . 19
1.3.3 Background on Psi-calculi . . . 19
1.4 Main results . . . 21

2. Non-interleaving systems as Psi-calculi instances . . . 23
2.1 Introduction . . . 23
2.2 Background on Psi-calculi . . . 25
2.2.1 Terms . . . 30
2.3 Representing event structures in Psi-calculi . . . 32
2.3.1 Background on event structures . . . 32
2.3.2 The encoding of prime event structures . . . 35
2.3.3 Action refinement . . . 39
2.4 DCR-graphs as Psi-calculi . . . 41
2.4.1 Background on DCR-graphs . . . 41
2.4.2 Encoding DCR-graphs . . . 44
2.4.3 Correlation between dcrPsi and eventPsi . . . 53
2.5 Conclusions and outlook . . . 57

3. Non-interleaving semantics for pi-calculus . . . 59
3.1 Introduction . . . 59
3.2 Causal Early Operational Semantics . . . 60
3.2.1 Pi-calculus syntax . . . 60
3.3 Correctness results . . . 66
3.4 Labelled asynchronous transition systems from early operational semantics of pi-calculus . . . 72
3.5 Conclusion and Related work . . . 84

4. Towards non-interleaving semantics for Psi-calculi . . . 85
4.1 Causality in Psi-calculi . . . 85
4.1.1 Main differences between guarded pi-calculus and the Psi-calculi . . . 88
4.1.2 Observing environmental-causality in Psi-processes . . . 91
4.1.3 Non-interleaving semantics for Psi-calculi, a suggestion . . . 92
4.2 eventPsi and dcrPsi as sanity checks of environmental causality . . . 92
4.2.1 Event structure independence in Psi-calculi . . . 94

4.2.2 DCR-graph independence in Psi-calculi . . . 94
4.3 Conclusion and further work . . . 96

LIST OF FIGURES

1.1 Transition system from interleaving semantics . . . 9
1.2 Refining an interleaving semantic TS . . . 10
1.3 Proper refining . . . 10
1.4 Classic examples of event structures. . . . 12
1.5 Configurations and steps for event structures . . . 13
1.6 Example for action refinement . . . 14
1.7 Simple DCR-graph of a read-send process . . . 17
1.8 DCR-graph with added responses . . . 17
1.9 Message forwarder DCR-graph with possible infinite execution. . . . 18
1.10 Summary of the results in chapter 2 . . . 21

2.1 Summary of the results in this chapter . . . 24
2.2 Classic examples of event structures. . . . 33
2.3 Configurations and steps for event structures . . . 34
2.4 Example for action refinement . . . 40
2.5 Simple DCR-graph of a read-send process . . . 43
2.6 DCR-graph with added responses . . . 44
2.7 Message forwarder DCR-graph with possible infinite execution. . . . 44
2.8 Example of cancelling infinitely many different events coming from a loop unfolding. . . . 58

3.1 Non-interleaving vs. interleaving diamond . . . 61
3.2 Early non-interleaving operational semantics for pi-calculus . . . 62
3.3 Link causality exemplified in Ex.3.2.8 . . . 63

4.1 Simplified partial transition system to process in Example 4.1.1 . . . 86
4.2 DCR-graph G simulating multiple extruders . . . 88
4.3 choice labels only from (CHOICE) rule . . . 90
4.4 choice labels with (CHOICEHISTORY) and (CHOICE) rules . . . 90
4.5 Semantic rules for Psi-calculi treating the accumulation of extruders information in the histories. . . 93
4.6 DCR-graphs of arguably non-orthogonal independent events . . . 95

1. INTRODUCTION

1.1 Introduction

The PhD project resulting in this thesis was funded by the research project CompArt[1], focusing on providing a foundation for the description and formal reasoning of adaptable, distributed and mobile computational artefacts in cooperative work settings under regulative control.

Examples of such computational artefacts are workflow management systems which are examples of process aware information systems [RW12]. The processes encoded in workflow management systems are often regulated by laws and other compliance rules. These processes need to be able to be adapted as the laws change, in order for the behaviour to stay compliant. Many cooperative work practices, for instance at hospitals, are distributed among different locations and require the sharing of resources. In this thesis we explore mathematical models that encompass all these aspects.

Instead of inventing a new notation that encompass these aspects, we choose to start with Psi-calculi [BJPV11]. Psi-calculi is a metalanguage that constitute a parametric framework for nominal process calculi, where constraint based process calculi and process calculi for mobility can be defined as instances. Psi-calculi semantics originally take an interleaved approach to concurrency, which means that the semantics does not represent concurrency and distribution at all. This thesis takes the first steps towards the definitions of non-interleaving semantics in Psi-calculi, where concurrency and distribution is observable in the semantics and opens for supporting adaptation by refinement [vGG01]. In addition, non-interleaving semantics allows reducing the impact of state space explosion [Val90, GW91, God96a, God96b]. The main contributions of this thesis are:

- Representation of non-interleaving models as instances of Psi-calculi.
- Non-interleaving operational semantics for the early pi-calculi.
- Identifying the obstacles that need to be overcome for non-interleaving semantics for the Psi-calculi.

In this first chapter we introduce what non-interleaving semantics is compared to interleaving semantics, provides the main definitions for the models we will be working with in the rest of the thesis, and finally go through the main results we have achieved during this work.

In chapter 2 we explore the strength of the logical aspects that Psi-calculi have. This we do through creating Psi-calculi instances of event structures [WN95, Ch.8] and DCR-graphs [HM10,HMS12], both being constraint event based non-interleaving models. We show that we can mirror the moves done in the original model, in the instance for a process that corresponds to the original model. The results in this chapter has previously been published in [NJH16b].

In chapter 3 we, as a preliminary step to the full Psi-calculi, give non-interleaving operational semantics for the early pi-calculus [MPW92]. These semantics is building on the work of Mukund and Nielsen [MN92] where they give non-interleaving semantics for CCS, and that of Crafa, Varacca and Yoshida [CVY12] where they provide denotational semantics to the late pi-calculus. This chapter is an extension of our paper presented at LATA2017 [HJN17].

In chapter 4 we look at the state of achieving non-interleaving semantics for Psi-calculi. We discuss how the assertions, conditions and the enabling function in Psi-calculi generate a new causality we call *environmental causality*. How this environmental causality behaves, depends on how the Psi-calculi instance is instantiated. We also look into how the instances of Event Structures and DCR-graphs can be used as sanity checks for an independence relation that can be generated for the full Psi-calculi.

[1] Computational Artefacts (Compart), http://www.compart.ku.dk

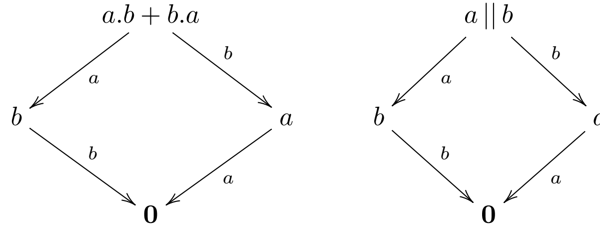


Fig. 1.1: Transition system from interleaving semantics

1.2 Semantic models of non-interleaving

1.2.1 Interleaving vs non-interleaving semantics

Interleaving semantics looks at parallelism as all possible interleavings of parallel events. This way to look at parallelism is a good approximation when one thinks about processes that run in a single thread. The problem with interleaving semantics is when we want to know what may have happened at the same time in a distributed system. Looking at an execution trace it becomes impossible to see the difference between processes that are sequential and those that are parallel.

We will look at this using an example simplified CCS processes. We have atomic actions refereed to by $a, b, c, d...$, sequential composition $a.b$ where a must happen before b, choice $P + P'$ meaning we have to choose either P or P' and parallel composition $P \,||\, P'$ where both P and P' can be executed at the same time. CCS also provide semantic reduction rules that say how and when an atomic action may be executed in some process P that then gives us a new process P'. A transition system(TS) is a collection of states and transitions between them. We formally define transition systems in the next subsection. From CCS we get that each process is a state, and the transitions between states are given by the actions to go from one process to another.

Take the following two processes "a then b or b then a" (written as $a.b + b.a$) and "a and in parallel b" (written as $a \,||\, b$). These two processes generate the transition systems seen in Figure 1.1. We can see that apart from the initial state both of these transition systems are identical. If we only look at the transitions and not the states they are observable equivalent.

Another issue with interleaving semantics is that they do not encompass the notion of refinement [vGG01]. Refinement means that we can take an atomic action and replace it with a more substantial process. We will look closer at refinement, particular action refinement of Event Structures, in Subsection 1.2.4. Assume that a is not an atomic action but rather the process "c then d" ($c.d$). We should then get the processes "c then d then b or b then c then d" ($c.d.b \vee b.c.d$) and "(c then d) and in parallel b" ($(c.d) \,||\, b$). From these we would be able to generate new TS respectively those in Figure 1.2, and Figure 1.3. If we would try to do refinement on the TS's we got in Figure 1.1 not knowing what the states are we would only get the TS in Figure 1.2. Assuming that we instead had a non-interleaving semantic we would have some additional information to the TS telling us what may happen concurrently, using this information we would get the TS in Figure 1.3 from the second TS in Figure 1.1 after the above refinement.

Other strengths of non-interleaved models compared to is that it helps with handling state space explosion [Val90, GW91, God96a, God96b]. We say that we have a state space explosion when a slightly larger process can generate, often exponentially, more states than a smaller one. The parallel composition of n concurrent tasks have n! different complete traces. However, if one knows that the tasks are independent, one can often avoid exploring all possible traces when analysing the system by using techniques known as partial order reduction [Val89, Pel93]. Also, some models like event structures and DCR-graphs avoid representing the traces explicitly and thus provide exponentially smaller models for concurrent systems than obtained using transition system models.

1.2.2 Transition systems and Asynchronous transition systems

Transition systems (TS) referred to in the previous section are the kind of system that most interleaving semantics generate.

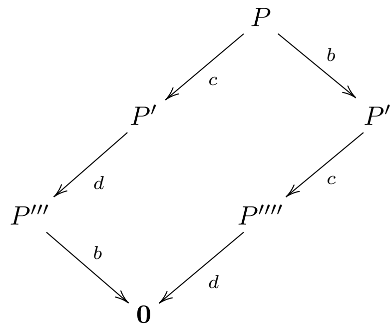


Fig. 1.2: Refining an interleaving semantic TS

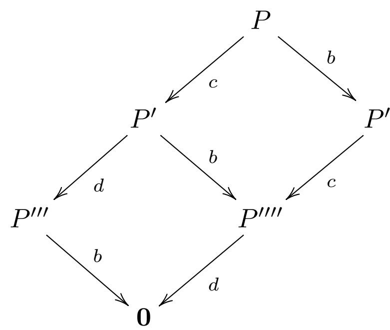


Fig. 1.3: Proper refining of "$a = c$ then d" in process $P = a$ and b

Definition 1.2.1: A transition system is a tuple $(S, i, E, \mathcal{T})$ where:

- S is the set of states
- i is an initial state such that $i \in S$
- E is a set of events
- $\mathcal{T}$ is the set of transitions being a subset of $S \times E \times S$.

A TS has no information about concurrency at all, only stating how one can get from one state to another, if possible, through a sequential execution.

Asynchronous Transition Systems (ATS) [Bed87, Shi85a] is an extension to TS which adds information about independence between events. If two events are independent, then they do not affect the ability for each other to be executed. When we can see that two events will not interfere with each other, we may claim that they could have happened concurrently, if they were enabled at the same time. This makes ATS a non-interleaving semantic as we can clearly see the difference between what must be sequential execution and what may be concurrent.

Definition 1.2.2: An *asynchronous transition system* is a tuple $ATS = (S, i, E, \mathcal{T}, I)$ where

- $(S, i, E, \mathcal{T})$ is a transition system with S the set of *states* and i an *initial state*, E a set of *events*, and $\mathcal{T} \subseteq S \times E \times S$ the *transition relation*;
- $I \subseteq E \times E$ is an irreflexive, symmetric independence relation, satisfying:
  1. $e \in E \Rightarrow \exists s, s' \in S : (s, e, s') \in \mathcal{T}$;
  2. $(s, e, s') \in \mathcal{T} \wedge (s, e, s'') \in \mathcal{T} \Rightarrow s' = s''$;
  3. $e_1 I e_2 \wedge \{(s, e_1, s_1), (s, e_2, s_2)\} \subseteq \mathcal{T} \Rightarrow \exists s_3 : \{(s_1, e_2, s_3), (s_2, e_1, s_3)\} \subseteq \mathcal{T}$;
  4. $e_1 I e_2 \wedge \{(s, e_1, s_1), (s_1, e_2, s_3)\} \subseteq \mathcal{T} \Rightarrow \exists s_2 : \{(s, e_2, s_2), (s_2, e_1, s_3)\} \subseteq \mathcal{T}$.

Example 1.2.3: Looking back at the examples from Figure 1.1 we get the transition systems $TS = (S, a.b + b.a, (a, b), \mathcal{T})$ and $TS' = (S', a \,||\, a, (a, b), \mathcal{T}')$. Here we have the same amount of states, events and transitions in both systems, actually the only difference between them is the initial state. While in this example it is possible to read what is independent from the initial state, sometimes we do not know how the states look like, the transition systems do not give us any more information regarding what could be concurrent.

Assume now that we have a method to identifying which events are independent of each other and thus generate an independence relation. Using this independence relation on the above processes we could generate the following two ATS: $(S, a.b + b.a, \{a, b\}, \mathcal{T}, \emptyset)$ and $(S', a \,||\, b, \{a, b\}, \mathcal{T}', \{(a, b)\})$ which clearly shows a difference between the process with only sequential execution and the one with parallel execution, in what independence relations we get.

In terms of refinement of processes, we have that it preserves independence between actions/events. That independence is preserved means that if a and b are independent and we refine a to be $c.d$, then both c and d are independent of b. The knowledge of what is independent and the requirement that refinement preserves independence makes refining the ATS gotten from the process $a \,||\, b$, (that is the second TS in Figure 1.1 added the independence relation $I = \{(a, b)\}$) gives us the ATS in Figure 1.3 with the independence relation $I' = \{(c, b), (d, b)\}$.

1.2.3 Event Structures

Event structures(ES) is a model of concurrency where we for this thesis follow the standard notation and terminology from [WN95, Ch.8].

Definition 1.2.4 (prime event structures): *A labelled prime event structure* over alphabet Act is a tuple $\mathcal{E} = (E, \leq, \sharp, l)$ where E is a possibly infinite set of events, $\leq \subseteq E \times E$ is a partial order (the *causality* relation) satisfying

1. *the principle of finite causes*, i.e.: $\forall e \in E : \{d \in E \mid d \leq e\}$ is finite,

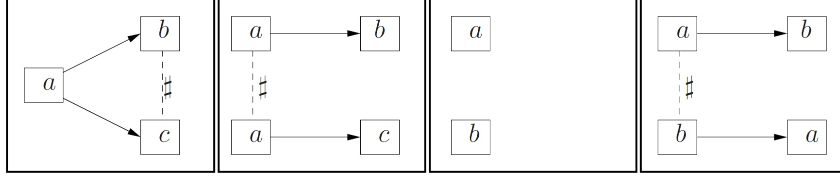


Fig. 1.4: Classic examples of event structures. Different boxes represent different events, possibly labelled the same. Arrows represent causality (not displaying those coming from the transitive closure); and dashed lines represent the symmetric conflicts (not displaying those coming from heredity).

and $\sharp \subseteq E \times E$ is an irreflexive, symmetric binary relation (the *conflict* relation) satisfying

2. *the principle of conflict heredity*, i.e., $\forall d, e, f \in E : d \leq e \wedge d \sharp f \Rightarrow e \sharp f$.

and $l : E \rightarrow Act$ is the labelling function. In the rest of the paper we only consider finite prime event structures, that is prime event strctures where the set of events E is finite. Denote by $\mathbb{E}$ the class of all finite prime event structures.

Intuitively, a prime event structure models a concurrent system by taking $d \leq e$ to mean that event d is a prerequisite of event e, i.e., event e cannot happen before event d has been done. A conflict $d \sharp e$ says that events d and e cannot both happen in the same run. Compared to other models of concurrency like process algebras, event structures model systems by looking only at their events, and how these events relate to each other. The two basic relations considered by event structures are the *dependency* and the *conflict* relations. The conflict relation can be used to capture choices made by the system, since the execution of one event discards all other events in conflict with itself for the rest of the computation.

Labels can be understood as actions, with a wide and general meaning. Events are instances of actions, and an action can happen several times, thus as different events. The same action can also happen in different components running in parallel, giving rise to *autoconcurrency* [DGK00]. Actions are important for observational equivalence, but not only them (see Example 1.2.8).

Example 1.2.5: In Figure 1.4 we pictured four simple examples of finite event structures (taken from [vGV97, Fig.4]). We illustrate events as boxes containing their labels, the dependency relation by arrows, and the conflict relation by dashed lines with a $\sharp$ sign. In the left-most event structure we have three events, where the events labelled b and c depend on the event labelled a and are in conflict with each other. This is a standard branching point which could be specified in a simple CCS notation as $a; (b + c)$. In the second event structure we have two (conflicting) events, both labelled with a, and two events labelled b and c which depend on the first and the second a-labelled event respectively. Because of the principle of conflict hereditary, b is in conflict with the lower a-labelled event and similarly, c is in conflict with the upper a-labelled event (but the conflict relations are in this case usually not explicitly illustrated). In CCS notation this could be $a; b + a; c$. In the third event structure we have just two events without any explicit or inherited relation, which means that they are *concurrent* (e.g., $a||b$), as made precise below. The last event structure is similar to the second, except that it offers two conflicting paths with the label a followed by b or b followed by a respectively.

Definition 1.2.6 (concurrency): *Causal independence (concurrency)* between events is defined in terms of the above two relations as

$$d||e \triangleq \neg(d \leq e \vee e \leq d \vee d \sharp e).$$

This definition captures the intuition that two events are concurrent when there is no causal dependence between the two and, moreover, they are not in conflict.

From the definition it follows that only the two events in the third event structure in Figure 1.4 are concurrent.

The behaviour of an event structure is described by subsets of events that happened in some (partial) run of the system being modelled. This is called a *configuration* of the event structure, and *steps* can be defined between configurations.

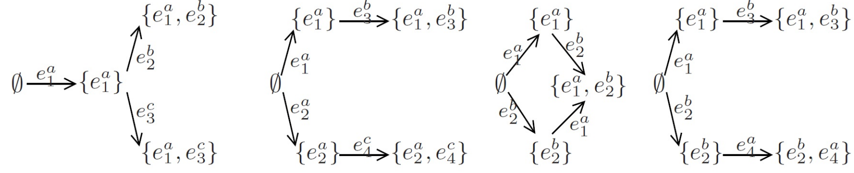


Fig. 1.5: Configurations and steps for event structures in Figure 1.4. The labels of the events are shown as superscripts.

Definition 1.2.7 (configurations): Define a *configuration* of an event structure $\mathcal{E} = (E, \leq, \sharp)$ to be a finite subset of events $C \subseteq E$ that respects:

1. *conflict-freeness*: $\forall e, e' \in C : \neg(e \sharp e')$ and,
2. *downwards-closure*: $\forall e, e' \in E : e' \leq e \wedge e \in C \Rightarrow e' \in C$.

We denote the set of all configurations of some event structure by $\mathbb{C}_{\mathcal{E}}$.

Conflict-freeness is saying that no two conflicting events can happen in one run. This also says that once an event is discarded it can never be executed on the current run. This is similar to how the semantics of the choice operator in process algebras is defined (see rule (SUM) in Definition 3.3.2) where all other branches of the choice are discarded once a step is taken. The downwards-closure says that all the dependencies of an executed event (i.e., which is part of a configuration) must have been executed also (on this same run).

Note in particular that $\emptyset$ is a configuration (i.e., the root configuration) and that any set $\lceil e \rceil \triangleq \{e' \in E \mid e' \leq e\}$ is also a configuration determined by the single event e. Events determine steps between configurations in the sense that $C \xrightarrow{e} C'$ whenever C, C' are configurations, $e \notin C$, and $C' = C \cup \{e\}$.

Example 1.2.8: For the examples from Figure 1.4 we get the configurations and steps depicted in Figure 1.5.

One may note that if only the paths of labels are observed, the two first event structures are indistinguishable, but if the branching structure is observed, i.e. by a bisimulation equivalence, they are distinguishable. One may also note that if the paths of labels are observed and even if branching time is observed, the two latter event structures are indistinguishable, but if concurrency is observed, i.e. by a history-preserving bisimulation equivalence [DDM88, vGG01], they are distinguishable.

Remark 1.2.9: It is known (see e.g., [WN95, Prop.18]) that prime event structures are fully determined by their sets of configurations, i.e., the relations of causality, conflict, and concurrency can be recovered only from the set of configurations $\mathbb{C}_{\mathcal{E}}$ as follows:

1. $e \leq e'$ iff $\forall C \in \mathbb{C}_{\mathcal{E}} : e' \in C \Rightarrow e \in C$;
2. $e \sharp e'$ iff $\forall C \in \mathbb{C}_{\mathcal{E}} : \neg(e \in C \wedge e' \in C)$;
3. $e || e'$ iff $\exists C, C' \in \mathbb{C}_{\mathcal{E}} : e \in C \wedge e' \notin C \wedge e' \in C' \wedge e \notin C' \wedge C \cup C' \in \mathbb{C}_{\mathcal{E}}$.

It is also known (see e.g., [WN95, Sec.8] for prime event structures or [vGG01, Sec.4] for the more general event structures of [Win87]) that there is no loss of expressiveness when working with finite, instead of infinite configurations. An infinite configuration can be obtained from infinite union of finite configurations coming from an infinite run.

For some event e we denote by $\leq e = \{e' \in E \mid e' \leq e\}$ the set of all events which are conditions of e (which is the same as the notation $\lceil e \rceil$ from [WN95], but we prefer to use the above so to be more consistent with similar notations we use in the rest of this thesis for similar sets defined for DCRs too), and $\sharp e = \{e' \in E \mid e' \sharp e\}$ those events in conflict with e. We denote by $<e = \leq e \setminus \{e\}$ the non-trivial conditions of e, i.e., excluding itself.

From [SNW96] we know that an ATS unfolds to a prime event structure, and that from an event structure we can generate an ATS with transitions between configurations.

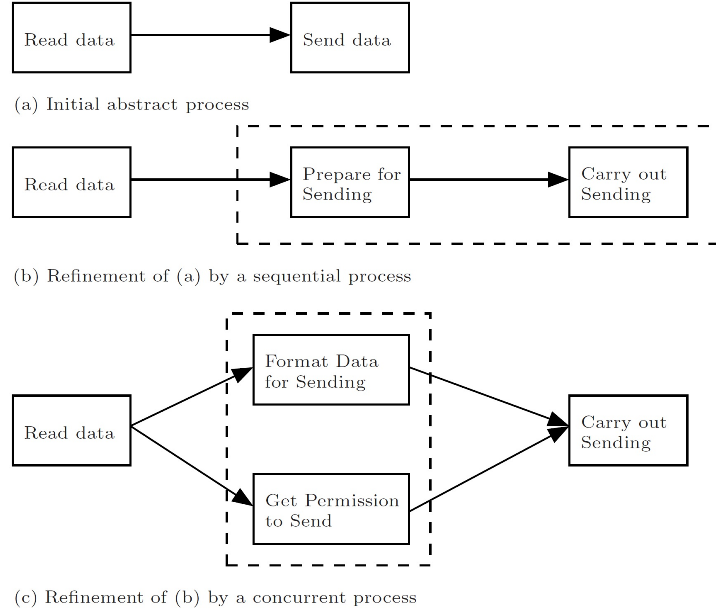


Fig. 1.6: Example for action refinement from [vGG01].

1.2.4 Action refinement

Below we recall *action refinement* for labelled event structures [vGG01]. The intuition of action refinement is to be able to give actions (which are thought of as possible abstractions) more structure, by replacing every event labelled by a particular action with a finite, conflict free event structure (with events possibly labelled by other actions). For example one action can be refined into a sequence of actions, or in general any deterministic finite concurrent process.

for some labelling function $l : E \to Act$, and as usual write e^a for an event (e, a).

A *refinement function* $ref : Act \to \mathbb{E}_{\not\sharp}$ is then a function from the set of actions of event structures (denoted by *Act*) to conflict-free event structures (i.e., the conflict relation is empty) denoted by $\mathbb{E}_{\not\sharp}$. The function *ref* is considered as a given function to be used in the *refinement operation* denoted by **ref**.

Definition 1.2.10 (refinement for prime event structures):

For an event structure $\mathcal{E}$ with events labelled by $l : E \to Act$ a function $ref : Act \to \mathbb{E}_{\not\sharp}$ is called a *refinement function* (for prime event structures) iff $\forall a \in Act : ref(a)$ is a non-empty, finite and conflict-free labelled prime event structure.

For $\mathcal{E} \in \mathbb{E}$ and *ref* a refinement function, let $\mathbf{ref}(\mathcal{E}) \in \mathbb{E}$ be the prime event structure defined by:

- $E_{\mathbf{ref}(\mathcal{E})} := \{(e, e') | e \in E_{\mathcal{E}}, e' \in E_{ref(l_{\mathcal{E}}(e))}\}$, where $E_{ref(l_{\mathcal{E}}(e))}$ denotes the set of events of the event structure $ref(l_{\mathcal{E}}(e))$,
- $(d, d') \leq_{\mathbf{ref}(\mathcal{E})} (e, e')$ iff $d \leq_{\mathcal{E}} e$ or $(d = e \wedge d' \leq_{ref(l_{\mathcal{E}}(d))} e')$,
- $(d, d') \sharp_{\mathbf{ref}(\mathcal{E})} (e, e')$ iff $d \sharp_{\mathcal{E}} e$,
- $l_{\mathbf{ref}(\mathcal{E})}(e, e') := l_{ref(l_{\mathcal{E}}(e))}(e')$.

Example 1.2.11: Figure 1.6(a) provides an illustrative example (taken from [vGG01]) of action refinement applied to a process which receives and sends data. We may think of giving more details to the sending event by refining

it with a sequential process which first prepares the data and then carries out the actual sending. This refined process is shown in Figure 1.6(b). In turn, Figure 1.6(c) shows a further refinement, where the preparation of data is refined by a process which in parallel formats the data and asks permission to send.

Remark 1.2.12: Action refinement as presented here was introduced by Wirth [Wir71] under the name of *stepwise refinement* and is quite different than the more recent notion of refinement in process algebras where the refined process is seen as an *implementation* of the abstract one. In these settings usually refinement is seen as an inclusion-like relation of the behaviours, such as trace inclusion or simulation.

1.2.5 DCR-graphs

Dynamic Condition Response graphs (DCR-graphs) [HM10, HMS12] is a model of concurrency which generalises event structures in two dimensions: Firstly, it allows finite models of (regular) infinite behaviour, while retaining the possibility of infinite models. The finite models are regular in the automata-theoretic sense, i.e. they (if concurrency is ignored) capture exactly the languages that are the union of a regular and an omega-regular language [DHS15a]. Finite DCR-graphs have found applications in practice for the description, implementation and automated verification of flexible workflow systems [Sla15, SMHM13]. Infinite DCR-graphs allow for representation of non-regular behaviour and denotational semantics. Secondly, the DCR-graphs model provides an event-based notion of acceptance criteria for both finite and infinite computations in terms of scheduled responses. We follow the notations for DCR-graphs from [HM10, HMS12].

Definition 1.2.13 (DCR Graphs): We define a *Dynamic Condition Response Graph* to be a tuple $\mathcal{D} = (E, M, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%, L, l)$ where

1. E is a set of events,
2. $M \in 2^E \times 2^E \times 2^E$ is the initial marking,
3. $\rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\% \subseteq E \times E$ are respectively called the *condition*, *response*, *milestone*, *include*, and *exclude* relations,
4. $l : E \rightarrow L$ is a *labelling function* mapping events to labels taken from L.

For any relation $\rightarrow \in \{\rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%\}$, we use the notation $e\rightarrow$ for the set $\{e' \in E \mid e \rightarrow e'\}$ and $\rightarrow e$ for the set $\{e' \in E \mid e' \rightarrow e\}$.

A *marking* $M = (Ex, Re, In)$ represents a state of the DCR graph. One should understand Ex as the set of *executed* events, Re the set of scheduled *response* events[2] that must happen sometime in the future or become excluded for the run to be accepting (see Definition 1.2.19), and In the set of currently *included* events. The five relations impose constraints on the events and dictate the dynamic inclusion and exclusion of events.

Intuitively, the condition relation $e \rightarrow\!\!\bullet\, e'$ requires the event e to have happened (at least once) or currently be excluded in order for e' to happen. The response relation $e \bullet\!\!\rightarrow e'$ means that if the event e happens, then the event e' becomes scheduled as a response. The milestone relation $e \rightarrow\!\!\diamond\, e'$ imposes the constraint that e' cannot happen as long as e is a scheduled response and included. Finally, the exclusion and inclusion relations generalize the conflict relation from event structures. An event e that excludes another event e' can be thought as being in (one-sided) conflict; but another event may include e' again, thus making the previous conflict only *transient*.

An event is thus *enabled* if it is included, all its included preconditions have been executed, and none of the included events that are milestones for it are scheduled responses. In particular, an event can happen an arbitrary number of times as long as it is enabled. We express the enabling condition formally as follows.

Definition 1.2.14 (enabling events): For a DCR graph $\mathcal{D} = (E, M, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%)$ with an initial marking $M = (Ex, Re, In)$, we say that an *event* $e \in E$ *is enabled in* M, written $M \vdash e$, iff

$$e \in In \ \wedge\ (In \cap \rightarrow\!\!\bullet e) \subseteq Ex \ \wedge\ (In \cap \rightarrow\!\!\diamond e) \subseteq (E \setminus Re).$$

[2] similar to the notion of restless events in [Win80, ch.6.4.].

Having defined when events are enabled, we can define an event labelled transition semantics for DCR-graphs. Since the execution of an event only changes the marking, we define the transition relation between the markings of a given DCR-graph and regard the marking M given in the DCR-graph as the initial marking.

Definition 1.2.15 (transitions): The behaviour of a DCR-graph is given through *transitions between markings* done by executing enabled events. The result of the execution in a DCR-graph $\mathcal{D} = (E, M_0, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%)$ from marking $M = (Ex, Re, In)$ of an enabled event $M \vdash e$ results in the new marking

$$M' \stackrel{def}{=} (Ex \cup \{e\}, (Re \setminus \{e\}) \cup e\bullet\!\rightarrow, (In \setminus e\rightarrow\!\%) \cup e\rightarrow\!+).$$

and is written as the e-labelled transition $M \xrightarrow{e} M'$. The (interleaving) semantics of the DCR-graph is then defined as the event-labelled transition system with markings as states and M_0 the initial state.

Using this definition of transitions will we get a TS. By adding the definitions of effect-orthogonal, cause-orthogonal and independence between events from [DHS15b] can we then generate an ATS.

Definition 1.2.16: We say that events $e \neq f$ of a DCR-graph G are *effect-orthogonal* iff

1. no event included by e is excluded by f and vice versa, and
2. e requires a response from some g iff f does.

Definition 1.2.17: Events e, f of a DCR-graph G are *cause-orthogonal* iff

1. neither event is a condition for the other;
2. neither event includes the other, and
3. neither event includes or excludes a condition of the other.

From these two definitions we get

Definition 1.2.18: Given a DCR-graph G, we say that events e, f are *independent* if they are both effect- and cause-orthogonal. We write I_G for the independence relation induced by a DCR-graph G.

We can now define (possibly infinite) runs of DCR-graphs and the acceptance criteria formally. As stated above, every event scheduled as response must either happen or be excluded in the future, in order for the run to be accepting. An event is no longer scheduled as a response after it has happened, unless it is related to itself by a response relation.

Definition 1.2.19 (accepting runs [HM10]): A *run* of a DCR-graph with initial marking M_0 is a (possibly infinite) sequence of transitions $M_i \xrightarrow{e_i} M_{i+1}$, with $0 \leq i < k$, $k \in \mathbb{N} \cup \{\omega\}$, and $M_i = (Ex_i, Re_i, In_i)$. A run of a DCR graph is *accepting* (or completed) if it holds that

$$\forall i \geq 0, e \in Re_i. \exists j \geq i : (e = e_j \vee e \notin In_j)$$

In words, a run of a DCR-graph is accepting if no event scheduled as an response is included and pending forever without happening, i.e. it must either eventually happen on the run or become excluded.

It is worth noting that the labelling function on events adds the possibility of non-determinism by taking the language of a DCR-graph to be the sequences of labels of events (abstracting from the events) of accepting runs. This extra level of labelling increases the expressive power of finite DCR graphs, which have been shown to capture exactly the languages that are the union of a regular and an omega-regular language [DHS15a]. In contrast, by following an encoding along the lines of [RHT08] unlabelled, finite DCR-graphs can be represented by Linear-time Temporal Logic (LTL), which is known to be strictly less expressible than omega-regular languages.

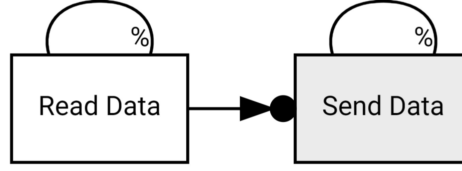


Fig. 1.7: Simple DCR-graph of a read-send process

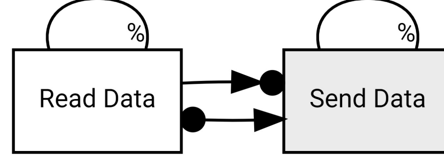


Fig. 1.8: DCR-graph with added responses

As given by the mapping in Definition 1.2.20 below, prime event structure can be seen as a special case of a DCR-graph (see [HM10, Prop.1&3] for details) where the exclusion relation (capturing the conflict relation) is reflexive and symmetric, the condition relation (capturing the causality relation) is irreflexive and transitive, and the include, response, and milestone relations are empty. The initial marking has no executed events, no scheduled responses and all events are included. Hereto comes of course the two conditions on the causality and conflict relation of event structures, i.e. finite causes and hereditary conflict. The reflexivity of the exclusion relation and emptiness of the inclusion relation imply that events can be executed at most once.

Definition 1.2.20 (prime event structures as DCR graphs [HM10]): Define a mapping **dcr** which takes an event structure $\mathcal{E} = (E, \leq, \sharp, l)$ and returns its presentation as a DCR-graph $(E, M, <, \emptyset, \emptyset, \emptyset, \sharp \cup \{(e,e) \mid e \in E\})$ with the marking $M = (\emptyset, \emptyset, E)$.

Example 1.2.21: Consider the small DCR-graph $\mathcal{D}$ shown in Figure 1.7, corresponding to the event structure of Figure 1.6(a) which was representing a process of first reading data ($\textsc{r}$) and then sending data ($\textsc{s}$). The DCR-graph is formalised as

$$\mathcal{D}_1 = (\{\textsc{r}, \textsc{s}\}, (\emptyset, \emptyset, \{\textsc{r}, \textsc{s}\}), \{(\textsc{r}, \textsc{s})\}, \emptyset, \emptyset, \emptyset, \{(\textsc{r}, \textsc{r}), (\textsc{s}, \textsc{s})\}).$$

Here we can see that each event removes itself from the included set when it happens, and that for $\textsc{s}$ to happen its prerequisite $\textsc{r}$ must happen. This corresponds to the causality relation in the event structure in Figure 1.6(a).

In DCR-graphs is possible to demand that if we read some data we will eventually send this data. This is modelled in the DCR-graph of Figure 1.8 formalised as

$$\mathcal{D}_2 = (\{\textsc{r}, \textsc{s}\}, (\emptyset, \emptyset, \{\textsc{r}, \textsc{s}\}), \{(\textsc{r}, \textsc{s})\}, \{(\textsc{r}, \textsc{s})\}, \emptyset, \emptyset, \{(\textsc{r}, \textsc{r}), (\textsc{s}, \textsc{s})\}),$$

where we added a response relation from $\textsc{r}$ to $\textsc{s}$. This means that a run is only accepting if any $\textsc{r}$ is eventually followed by an $\textsc{s}$, e.g., the empty run and the run $\textsc{r}.\textsc{s}$ are accepting, while the run consisting of the single event $\textsc{r}$ is not.

The examples above only allow each event to happen once. However, as exemplified below, in DCR graphs the set of events needed for an event to be enabled can change during the run, as events are included or excluded. Moreover, the conflict in DCR-graphs is not permanent as is the case with event structures. Conflict in DCR graphs is *transient* since an event can be included and later excluded during a run. So, already at the conflict and causality relations, the DCR-graphs depart from event structures in a non-trivial manner.

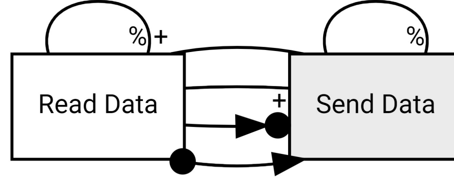


Fig. 1.9: Message forwarder DCR-graph with possible infinite execution.

Example 1.2.22: A message forwarding machine, where the events can happen several times, but alternating, can be represented by the DCR-graph in Figure 1.9 formalised as

$$\mathcal{D}_3 = (\{\text{R}, \text{S}\}, (\emptyset, \emptyset, \{\text{R}, \text{S}\}), \{(\text{R}, \text{S})\}, \{(\text{R}, \text{S})\}, \emptyset, \{(\text{R}, \text{S}), (\text{S}, \text{R})\}, \{(\text{R}, \text{R}), (\text{S}, \text{S})\}).$$

Each time one of the two events happens it excludes itself but includes the other event, modelling the alternation between reading and sending. We still have that if R happens, eventually S must happen for the run to be accepting, but we make no requirements on how many times the events can happen, so the unique infinite execution and every finite execution of even length will be accepting.

1.3 Process models

1.3.1 Pi-calculus

Pi-calculus was originally presented by Robin Milner in [MPW92] as a model of mobile processes. Pi-calculus has an interleaving semantically approach to concurrency with channels transmitting names around the process. While from the pi-process one can see that two transitions may be concurrent from its structure like in the process $\overline{a}\langle n\rangle \,||\, \overline{n}\langle m\rangle$ where both transitions are independent and should be allowed to happen concurrently. This is though for pi-calculus not enough to see what can be concurrent, we also need to know what names are being sent around in order to properly see what is dependent as seen in [CVY12]. Building on the first process we have the process $(\nu n)(\overline{a}\langle n\rangle \,||\, \overline{n}\langle m\rangle)$ where the transition coming from $\overline{n}\langle m\rangle$ has to wait for the transition coming from $\overline{a}\langle n\rangle$ because of the scoping of name n. We will return to the properties of dependencies in pi-calculus in Chapter 3.

Bellow we recall the syntax for pi-calculus, following a style similar to that in [Qua99].

Definition 1.3.1 (Pi-calculus syntax): The set of pi-calculus processes **Proc**, ranged over by $P, Q, \ldots$, are defined by the grammar

$$P ::= \overline{a}\langle n\rangle.P \;\mid\; \underline{a}(x).P \;\mid\; P + Q \;\mid\; (\nu a)P \;\mid\; P \,||\, Q \;\mid\; !P \;\mid\; \mathbf{0}$$

providing constructs for, respectively: output, input, non-deterministic choice, restriction of name a, parallel composition, replication and the empty/trivial process and assuming an infinite set of names $\mathcal{N}$.

We make a habit of using $a, b, ...$ for names that are meant to indicate channels/links for communication, $m, n, ...$ for names that are being transmitted over channels, and $x, y, z, ...$ when we intend these names to be substituted.

Below we provide the action labels, ranged over by α, for the transition semantics. The operational semantics is standard and we recall the presentation given by [Qua99] in Definition 1.3.4.

Definition 1.3.2 (action labels): The semantics of pi-calculus uses action labels on transitions.

τ	Communication between parallel components;
$\overline{a}\langle n\rangle$	Output of name n on channel a when n is not under a restriction;
$\overline{a}(n)$	Output of name n on channel a when n is under a restriction;
$\underline{a}(n)$	Input of name n on channel a.

Notation 1.3.3: For a process P we denote by $n(P)$ the set of all names appearing in P, by $bn(P)$ those names that are restricted/bound by a (νn) or input action $\underline{a}(x)$, and by $fn(P)$ those names that are not bound (or free). For the particular process $(\nu n)\underline{a}(x)$ we have $n((\nu n)\underline{a}(x)) = \{n, a, x\}$, $bn((\nu n)\underline{a}(x)) = \{n, x\}$, and $fn((\nu n)\underline{a}(x)) = \{a\}$. For an action label α we use the same notation for similar notions. In particular, for an output action the bound name is that appearing under the (ν); e.g., $bn(\overline{a}(n)) = n$ whereas $bn(\overline{a}\langle n\rangle) = \emptyset$, and $fn(\overline{a}\langle n\rangle) = \{a, n\}$.

Definition 1.3.4 (Pi-calculus semantic rules): Denote by $\rightarrow_\pi$ the transitions obtained with the standard pi-calculus semantics [Qua99] over the pi-calculus syntax from Definition 1.3.1. We give these structural operational rules here (ignoring their tau-rule and match-rule, and omitting the symmetric versions of the (COM), (CLOSE) and (PAR) rule).

$$\frac{n \notin fn((\nu x)P)}{\underline{a}(x).P \xrightarrow{\underline{a}(n)}_\pi P[x := n]}\ (\textsc{inp}) \qquad \frac{}{\overline{a}\langle n\rangle.P \xrightarrow{\overline{a}\langle n\rangle}_\pi P}\ (\textsc{out})$$

$$\frac{P \xrightarrow{\alpha}_\pi P'}{P + Q \xrightarrow{\alpha}_\pi P'}\ (\textsc{sum}) \qquad \frac{P \xrightarrow{\alpha}_\pi P' \qquad bn(\alpha) \notin n(Q)}{P \,||\, Q \xrightarrow{\alpha}_\pi P' \,||\, Q}\ (\textsc{par})$$

$$\frac{P \,||\, !P \xrightarrow{\alpha}_\pi P'}{!P \xrightarrow{\alpha}_\pi P'}\ (\textsc{bang}) \qquad \frac{P \xrightarrow{\alpha}_\pi P' \qquad b \notin n(\alpha)}{(\nu b)P \xrightarrow{\alpha}_\pi (\nu b)P'}\ (\textsc{res})$$

$$\frac{P \xrightarrow{\overline{a}\langle n\rangle}_\pi P' \qquad Q \xrightarrow{\underline{a}(n)}_\pi Q' \qquad n \notin n(Q)}{P \,||\, Q \xrightarrow{\tau}_\pi P' \,||\, Q'}\ (\textsc{com}) \qquad \frac{P \xrightarrow{\overline{a}(n)}_\pi P' \qquad Q \xrightarrow{\underline{a}(n)}_\pi Q' \qquad n \notin n(Q)}{P \,||\, Q \xrightarrow{\tau}_\pi (\nu n)(P' \,||\, Q')}\ (\textsc{close})$$

$$\frac{P \xrightarrow{\overline{a}\langle n\rangle}_\pi P' \qquad n \neq a}{(\nu n)P \xrightarrow{\overline{a}(n)}_\pi P'}\ (\textsc{open})$$

1.3.2 Psi-calculus

1.3.3 Background on Psi-calculi

Psi-calculus [BJPV11] has been developed as a framework for defining nominal process calculi, like the many variants of the pi-calculus [MPW92]. The Psi-calculi framework is based on nominal datatypes, with [BJPV11, Sec.2.1] giving an introduction to nominal sets used in Psi-calculi. We will not expand much on nominal datatypes in this thesis, but refer the reader to the book [Pit13] which contains a thorough treatment of both the theory behind nominal sets as well as various applications (e.g., see [Pit13, Ch.8] for nominal algebraic datatypes). We expect, though, some familiarity with notions of algebraic datatypes and term algebras.

The Psi-calculi framework is parametric; instantiating the parameters accordingly, one obtains an *instance of Psi-calculi*, like the pi-calculus, or the cryptographic spi-calculus. These parameters are:

$\mathbf{T}$	terms (data/channels)
$\mathbf{C}$	conditions
$\mathbf{A}$	assertions

which are nominal datatypes not necessarily disjoint; together with the following operators:

$\dot{\leftrightarrow}\ :\ \mathbf{T} \times \mathbf{T} \rightarrow \mathbf{C}$	channel equality
$\otimes\ :\ \mathbf{A} \times \mathbf{A} \rightarrow \mathbf{A}$	composition of assertions
$\mathbf{1}\ \in\ \mathbf{A}$	minimal assertion
$\vdash\ \subseteq\ \mathbf{A} \times \mathbf{C}$	entailment relation

Intuitively, terms can be seen as generated from a signature, as in term algebras; the conditions and assertions can be those from first-order logic; the minimal assertion being top/true, entailment the one from first-order logic, and composition taken as conjunction. We will shortly exemplify how pi-calculus is instantiated in this framework. The operators are usually written infix, i.e.: $M \dot{\leftrightarrow} N$, $\Psi \otimes \Psi'$, $\Psi \vdash \varphi$.

The above operators need to obey some natural requirements, when instantiated. Channel equality must be symmetric and transitive. The composition of assertions must be associative, commutative, and have **1** as unit; moreover, composition must preserve equality of assertions, where two assertions are considered equal iff they entail the same conditions (i.e., for $\Psi, \Psi' \in \mathbf{A}$ we define the equality $\Psi \simeq \Psi'$ iff $\forall \varphi \in \mathbf{C} : \Psi \vdash \varphi \Leftrightarrow \Psi' \vdash \varphi$).

The intuition is that assertions will be used to assert about the environment of the processes. Conditions will be used as guards for guarded (non-deterministic) choices, and are to be tested against the assertion of the environment for entailment. Terms are used to represent complex data communicated through channels, but will also be used to define the channels themselves, which can thus be more than just mere names, as in pi-calculus. The composition of assertions should capture the notion of combining assumptions from several components of the environment.

The syntax for building Psi-process is the following (Psi-processes are denoted by the $P, Q, \ldots$; terms from $\mathbf{T}$ by $M, N, \ldots$;):

$\mathbf{0}$	Empty/trivial process
$\overline{M}\langle N \rangle.P$	Output
$\underline{M}\langle (\lambda \tilde{x}) N \rangle.P$	Input
$\mathbf{case}\ \varphi_1 : P_1, \ldots, \varphi_n : P_n$	Conditional choice
$(\nu a)P$	Restriction of names a
$P \,\|\, Q$	Parallel composition
$!P$	Replication
$(\!\|\Psi\|\!)$	Assertions

The empty process has the same behaviour as, and thus can be modelled by, the trivial assignment $(\!|\mathbf{1}|\!)$.

The input and output processes are as in pi-calculus only that the channel objects M can be arbitrary terms. In the input process the object $(\lambda \tilde{x})N$ is a pattern with the variables $\tilde{x}$ bound in N as well as in the continuation process P. Intuitively, any term message received on M must match the pattern N for some substitution of the variables $\tilde{x}$. The same substitution is used to substitute these variables in P after a successful match. The traditional pi-calculus input $a(x).P$ would be modelled in Psi-calculi as $\underline{a}\langle (\lambda x)x \rangle.P$, where the simple names a are the only terms allowed.

The case process behaves like one of the P_i for which the condition φ_i is entailed by the current environment assumption, as defined by the notion of *frame* which we present later. This notion of frame is familiar from the applied pi-calculus, where it was introduced with the purpose of capturing static information about the environment (or seen in reverse, the frame is the static information that the current process exposes to the environment). A particular use of case is as $\mathbf{case}\ \varphi : P$ which can be read as $\mathbf{if}\ \varphi\ \mathbf{then}\ P$. Another special usage of case is as $\mathbf{case}\ \top : P_1, \top : P_2$, where $\Psi \vdash \top$ is a special condition that is entailed by any assertion, like $a \dot{\leftrightarrow} a$; this use is mimicking the pi-calculus non-deterministic choice $P_1 + P_2$. Restriction, parallel, and replication are the standard constructs of pi-calculus.

Assertions $(\!|\Psi|\!)$ can float freely in a process (i.e., be put in parallel) describing assumptions about the environment. Otherwise, assertions can appear at the end of a sequence of input/output actions, i.e., these are the guarantees that a process provides after it finishes (on the same lines as in assume/guarantee reasoning about programs). Assertions are somehow similar to the active substitutions of the applied pi-calculus, only that assertions do not have computational behaviour, but only restrict the behaviour of the other constructs by providing their assumptions about the environment.

Example 1.3.5 (Pi-calculus as an instance): To obtain pi-calculus [MPW92] as an instance of Psi-calculi use the following, built over a single set of names $\mathcal{N}$:

$$\begin{array}{lcl}\mathbf{T} & \triangleq & \mathcal{N} \\ \mathbf{C} & \triangleq & \{a = b \mid a, b \in \mathbf{T}\} \\ \mathbf{A} & \triangleq & \{\mathbf{1}\} \\ \dot{\leftrightarrow} & \triangleq & = \\ \vdash & \triangleq & \{(\mathbf{1}, a = a) \mid a \in \mathbf{T}\}\end{array}$$

with the trivial definition for the composition operation. The only terms are the channel names $a \in \mathcal{N}$, and there is no other assertion than the unit. The conditions are equality tests for channel names, where the only successful tests are those where the names are equal. Hence, channel comparison is defined as just name equality.

Example 1.3.6: From the instance created in Example 1.3.5 one can obtain the polyadic pi-calculus [CM03] by adding tupling symbols t_n for tuples of arity n to $\mathbf{T}$, i.e.

$$\mathbf{T} = \mathcal{N} \cup \{t_n(M_1, ..., M_n) : M_1, ...M_n \in \mathbf{T}\}.$$

The polyadic output is to simply output the corresponding tuple of object names, and the polyadic input $a(b_1, ..., b_n).P$ is represented by a pattern matching

$$\underline{a}(\lambda b_1, ...b_n)t_n(b_1, ..., b_n).P.$$

Strictly speaking this allows nested tuples as well as tuples in the subject position of inputs and outputs. But these do not give rise to transitions because the definition of channel equality only applies to channel names, thus $M \dot{\leftrightarrow} M$ can be entailed by an assertion only when M is a name.

Psi-calculi maintain all the dependencies that exits in pi-calculus and obtain a new form of dependencies from the assertions and conditions. This is due to transitions leaving assertions behind that may enable or disable other transitions. More on this will be discussed in Chapter 4.

1.4 Main results

Figure 1.10 gives an overview of the main results from chapter 2. We give a representation of model of finite prime event structures [NPW79, Win87] as an instance of Psi-calculi in Section 2.3. The encoding function ESPSI, illustrated by the middle horizontal arrow in Figure 1.10 exploits the logic of Psi-calculi to represent the causality, independence and conflict relation of event structures. This allows us to prove that the representation respects the semantics up to concurrency diamonds and action refinement [vGG01]. Concretely, we define action refinement $\mathbf{ref}^{\Psi}$ on the eventPsi-processes, and prove in Sec. 2.3.3 that also action refinement is preserved by our translation, making the upper square diagram of Figure 1.10 commute.

We also identify the syntactic shape of Psi-processes which correspond to finite prime event structures. This last result can be seen as a characterisation of the Psi-processes for which the middle arrow in Figure 1.10 is one part of an equivalence.

$$\begin{array}{ccc}\mathbb{E} & \xrightarrow{\text{ESPSI}} & \text{eventPsi} \\ \uparrow{\scriptstyle\mathbf{ref}} & & \uparrow{\scriptstyle\mathbf{ref}^{\Psi}} \\ \mathbb{E} & \xrightarrow[\text{ESPSI}]{} & \text{eventPsi} \\ \downarrow{\scriptstyle\mathbf{dcr}} & & \downarrow{\scriptstyle\mathbf{emb}} \\ \mathbb{D} & \xrightarrow{\text{DCRPSI}} & \text{dcrPsi}\end{array}$$

Fig. 1.10: Summary of the results in chapter 2, where: $\mathbb{E}$ is the class of finite prime event structures, $\mathbb{D}$ is the class of DCR-graphs, eventPsi and dcrPsi are the two Psi-instances we define, **ref** is the refinement operation of [vGG01], $\mathbf{ref}^{\Psi}$ is the corresponding refinement operation we define for eventPsi processes, and **dcr** and **emb** are embeddings of event structures and eventPsi into DCR-graphs respectively dcrPsi.

As for event structures, we provide an encoding DCRPSI of DCR-graphs into the Psi-instance dcrPsi, shown in Figure 2.1 as the lower arrow. Again we identify the syntactic shape of those dcrPsi-processes which correspond exactly to DCR-graphs, i.e. characterises the dcrPsi-processes for which the mapping is part of an equivalence, and prove a bisimulation relation between the DCR-graphs semantics and their encoding for a naturally defined event-labelled transition system semantics of the encoding.

We end the section by showing that the encoding of DCR-graphs in dcrPsi is a conservative generalization of the encoding of finite event structures by showing that the lower square diagram in Figure 1.10 commutes, for the standard embedding **dcr** of event structures into DCR-graphs and a suitably defined, semantics preserving embedding **emb** of eventPsi-processes (corresponding to event structures) into dcrPsi-processes (corresponding to DCR-graphs).

In Chapter 3 we provide the first stable, non-interleaving operational semantics for the pi-calculus conservatively generalising the interleaving early operational semantics. The semantics is given as labelled asynchronous transition systems. We followed and conservatively generalised the the approach for CCS in [MN92] by capturing the link causalities in the pi-calculus processes by employing a notion of extrusion histories.

2. NON-INTERLEAVING SYSTEMS AS PSI-CALCULI INSTANCES

Psi-calculi constitute a parametric framework for nominal process calculi, where constraint based process calculi and process calculi for mobility can be defined as instances. We apply here the framework of Psi-calculi to provide a foundation for the exploration of declarative event-based process calculi with support for run-time refinement. We first provide a representation of the model of finite prime event structures as an instance of Psi-calculi and prove that the representation respects the semantics up to concurrency diamonds and action refinement. We then proceed to give a Psi-calculi representation of Dynamic Condition Response Graphs, which conservatively extends prime event structures to allow finite representations of (omega) regular finite (and infinite) behaviours and have been shown to support run-time adaptation and refinement.

2.1 Introduction

Software is increasingly controlling and supporting critical functions and processes in our society; from energy and transportation to finance, military and governmental processes. This makes the development of techniques for guaranteeing correctness of software systems increasingly important. At the same time, there is a growing need for the support of incremental development and adaptation of information systems, for the systems to be able to keep up with the changes in the physical and regulative context. This need is addressed in practice with the introduction of agile and continuous delivery software development methods and in theory by research in adaptable software systems and technology. Agility and adaptability, however, makes the non-trivial task of ensuring correctness of software systems even more difficult.

The present work is part of a more ambitious research goal pursued in the CompArt project.[1] The final aim is to provide a foundation for the description and formal reasoning of adaptable, distributed and mobile computational artefacts under regulative control.

A number of proposals have been made for concrete formal models and reasoning techniques for distributed and mobile systems supporting different kinds of adaptability (e.g. [BGPZ12, PGG$^+$15, GH05, BCH$^+$14, MHS13]). The plethora of models indicates that our understanding of the needs for such agile and adaptable computational artefacts is still developing. For this reason, we aim for a foundation that facilitates experimentation with different formal process calculi. Our focus on mobility, shared resources, regulative control and adaptability leads us to consider the formal meta process calculus of Psi-calculi [BJPV11]. Psi-calculi provides a general setting for the definition of process calculi for distributed and mobile processes, that generalises the seminal pi-calculus [MPW92] in two dimensions: *(i)* generalisation of channel names to nominal data structures [Pit13], i.e. general terms with a notion of local names; and *(ii)* a general logic for expressing constraints guarding actions. Examples of calculi encoded as Psi-calculi include the spi- and applied- pi-calculi [AG99, AF01] and the CC-pi [BM07, DFM$^+$05].

While the general nominal data structures and channel communication provide a foundation for expression of shared mobile resources, the constraint logic provides a foundation for declarative expression of control and regulations. Concretely, we show how two declarative, event-based process models for concurrency, i.e., the seminal prime event structures [NPW79, Win87] and the Dynamic Condition Response (DCR) Graphs [HM10], can be represented as Psi-calculi. We consider declarative event-based concurrent models for two reasons: Firstly, declarative models more naturally describe regulations, that is, rules governing processes. Secondly, they are well-behaved with respect to action refinement [Wir71, vGG01], which we see as a fundamental step towards supporting agile development and adaptation. Simply put, action refinement is the method of developing a system by starting with an abstract specification, and gradually refining its components (or actions) by providing more details. Thus an action can be

[1] Computational Artefacts (CompArt), http://www.compart.ku.dk

changed from being instantaneous to having structure, or duration. This should not be confused with the notion of refinement often found in process algebras where an implementation refines a specification by reducing the set of execution traces.

Figure 2.1 gives an overview of the main results from this chapter. After providing the necessary background on Psi-calculi in Section 2.2, we give a representation of model of finite prime event structures [NPW79, Win87] as an instance of Psi-calculi in Section 2.3. The encoding function ESPSI, illustrated by the middle horizontal arrow in Figure 2.1 exploits the logic of Psi-calculi to represent the causality, independence and conflict relation of event structures. This allows us to prove that the representation respects the semantics up to concurrency diamonds and action refinement [vGG01]. Concretely, we define action refinement $\mathbf{ref}^{\Psi}$ on the eventPsi-processes, and prove in Sec. 2.3.3 that also action refinement is preserved by our translation, making the upper square diagram of Figure 2.1 commute.

We also identify the syntactic shape of Psi-processes which correspond to finite prime event structures. This last result can be seen as a characterisation of the Psi-processes for which the middle arrow in Figure 2.1 is one part of an equivalence.

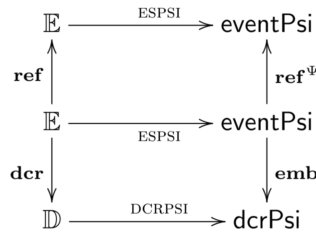


Fig. 2.1: Summary of the results in this chapter, where: $\mathbb{E}$ is the class of finite prime event structures, $\mathbb{D}$ is the class of DCR-graphs, eventPsi and dcrPsi are the two Psi-instances we define, **ref** is the refinement operation of [vGG01], $\mathbf{ref}^{\Psi}$ is the corresponding refinement operation we define for eventPsi processes, and **dcr** and **emb** are embeddings of event structures and eventPsi into DCR-graphs respectively dcrPsi.

Event structures are a denotational model and cannot provide finite representations of infinite behaviours. To accommodate finite representations of infinite behaviours, we proceed in Section 2.4 to consider Dynamic Condition Response graphs (abbreviated DCR-graphs or just DCRs) [HM10]. DCR-graphs are an event-based model of concurrency strictly generalizing event structures by permitting the events to happen more than once (as opposed to in event structures, where events happen at most once) and by refining the relations of dependency and conflict between events. This generalizes event structures in two ways: Firstly, DCR-graphs are a so-called system model allowing finite representations of infinite behaviours. Secondly, DCR-graphs allow representation of acceptance criteria for computations, making it possible to express both safety and liveness properties. We will not consider acceptance criteria in the present chapter. DCRs have been shown to support run-time adaptation [MHS13] and have been successfully used in industry to model and support flexible and adaptable business processes for knowledge workers [SMHM13, DHSM14].

As for event structures, we provide an encoding DCRPSI of DCR-graphs into the Psi-instance dcrPsi, shown in Figure 2.1 as the lower arrow. Again we identify the syntactic shape of those dcrPsi-processes which corresponds exactly to DCR-graphs, i.e. characterises the dcrPsi-processes for which the mapping is part of an equivalence, and prove a bisimulation relation between the DCR-graphs semantics and their encoding for a naturally defined event-labelled transition system semantics of the encoding.

We end the section by showing that the encoding of DCR-graphs in dcrPsi is a conservative generalization of the encoding of finite event structures by showing that the lower square diagram in Figure 2.1 commutes, for the standard embedding **dcr** of event structures into DCR-graphs and a suitably defined, semantics preserving embedding **emb** of eventPsi-processes (corresponding to event structures) into dcrPsi-processes (corresponding to DCR-graphs).

We end in Section 3.5 by concluding and outlining the path towards the final aim of this research, which is to explore nominal calculi for declarative, run-time adaptable mobile processes with shared resources, subject to both safety and liveness regulations.

This chapter extends [NPH14] by including new results, more examples, motivation and background. In particular, the background section on Psi-calculi has been considerably enlarged, including more definitions, examples and intuitions. The same was done for the backgrounds on DCR graphs and event structures and refinement. The results have more detailed proofs and include new results that complete the diagram of Figure 2.1 with the embedding of eventPsi into dcrPsi, as well as the syntactic restrictions for dcrPsi-processes that make the lower arrow part of an equivalence.

2.2 Background on Psi-calculi

Psi-calculi [BJPV11] have been developed as a framework for defining nominal process calculi, like the many variants of the pi-calculus [MPW92]. Instances of Psi-calculi have been made for applied pi-calculus [AF01] and for CC-pi [BM07, DFM+05] which can in turn capture probabilistic models. Typed Psi-calculi exist [Hüt11] as well as the related instance [Hüt14] for distributed pi-calculus [RH98, Hen07].

The Psi-calculi framework is based on nominal datatypes. We assume an infinite set of atomic *names* $\mathcal{N}$ ranged over by $a, b, \ldots$. Intuitively, names are symbols that can be statically scoped, as well as be subjected to substitution (which we define later). A nominal datatype is then constructed from a nominal set [Pit13], which is a set equipped with *name swapping* functions, written (ab), satisfying certain natural axioms such as $(ab)((ab)t) = t$. Intuitively, applying the name swapping (ab) changes a term t by replacing a with b and b with a. One main point is that even without having any particular syntax for constructing t we can define what it means for a name to "occur" in a term, i.e., when the term can be affected by a swapping that involves that name. The names occurring in this way in a term t constitute the *support* of t, written $n(t)$. In usual datatypes, without binders, we will have $a \notin n(t)$ if a does not occur syntactically in t. Whereas in the lambda calculus the support corresponds to the free names, since terms are identified up to alpha-equivalence. A function f is *equivariant* if $(ab)f(t) = f((ab)t)$ holds for all t. We can then define a nominal datatype formally as follows.

Definition 2.2.1 (nominal datatypes and substitutions): A *nominal datatype* is a nominal set together with a set of equivariant functions on it. Psi-calculi consider *substitution functions* that substitute terms for names. If t is a term of a datatype, $\tilde{a}$ is a sequence of names without duplicates, and $\tilde{T}$ is an equally long sequence of terms of possibly different datatypes, the *substitution* $t[\tilde{a} := \tilde{T}]$ is a term of the same datatype as t. The only formal requirements for substitutions are that a substitution is an equivariant function that satisfies two substitution laws:

1. if $\tilde{a} \subseteq n(t)$ and $b \in n(\tilde{T})$ then $b \in n(t[\tilde{a} := \tilde{T}])$
2. if $\tilde{b} \notin n(t)$ and $\tilde{b} \notin n(\tilde{a})$, then $t[\tilde{a} := \tilde{T}] = ((\tilde{b}\tilde{a})t)[\tilde{b} := \tilde{T}]$.

Law 1 says that a substitution does not lose names: any name b in the terms $\tilde{T}$ that substitute the names $\tilde{a}$ occurring in t must also appear in the resulting term after the substitution $t[\tilde{a} := \tilde{T}]$. Law 2 is a form of alpha-conversion for substitutions, where $\tilde{a}$ and $\tilde{b}$ have the same length, and $(\tilde{b}\tilde{a})$ swaps each name of $\tilde{a}$ with the corresponding name of $\tilde{b}$.

Definition 2.2.2 (parameters): The Psi-calculi framework is *parametric*; instantiating the parameters accordingly, one obtains an *instance of Psi-calculi*, like the pi-calculus, or the cryptographic spi-calculus. These parameters are:

$\mathbf{T}$	terms (data/channels)
$\mathbf{C}$	conditions
$\mathbf{A}$	assertions

which are nominal datatypes not necessarily disjoint; together with the following equivariant operators:

$\dot{\leftrightarrow} : \mathbf{T} \times \mathbf{T} \to \mathbf{C}$	channel equality
$\otimes : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$	composition of assertions
$\mathbf{1} \in \mathbf{A}$	minimal assertion
$\vdash \subseteq \mathbf{A} \times \mathbf{C}$	entailment relation

The operators are usually written infix, i.e.: $M \dot{\leftrightarrow} N$, $\Psi \otimes \Psi'$, $\Psi \vdash \varphi$.

Intuitively, terms can be seen as generated from a signature, as in term algebras [BN98]. We can think of the conditions and assertions like in first-order logic: the minimal assertion being top/true, entailment the one from first-order logic, and composition taken as conjunction. It is helpful to think of assertions and conditions as logical formulas, and the entailment relation as an entailment in logic; but allow the intuition to think of logics abstractly, not just propositional logic, so that assertions and conditions are used to express any logical statements, where the entailment defines when assertions entail conditions (do not restrict to only thinking of truth tables; e.g., in our encodings we will use an extended logic for sets, with membership, pairs, etc.). The intuition of entailment is that $\Psi \vdash \varphi$ means that given the information in Ψ, it is possible to infer φ. Two assertions are equivalent if they entail the same conditions.

Definition 2.2.3 (assertion equivalence): Two assertions are *equivalent*, written $\Psi \simeq \Psi'$, iff for all φ we have that $\Psi \vdash \varphi \iff \Psi' \vdash \varphi$.

The above operators need to obey some natural requirements, when instantiated.

Definition 2.2.4 (requisites on valid Psi-calculus parameters): The following properties must be satisfied by any Psi-instance.

Channel Symmetry:	$\Psi \vdash M \leftrightarrow N \implies \Psi \vdash N \leftrightarrow M$
Channel Transitivity:	$\Psi \vdash M \leftrightarrow N \wedge \Psi \vdash N \leftrightarrow L \implies \Psi \vdash M \leftrightarrow L$
Compositionality:	$\Psi \simeq \Psi' \implies \Psi \otimes \Psi'' \simeq \Psi' \otimes \Psi''$
Identity:	$\Psi \otimes \mathbf{1} \simeq \Psi$
Associativity:	$(\Psi \otimes \Psi') \otimes \Psi'' \simeq \Psi \otimes (\Psi' \otimes \Psi'')$
Commutativity:	$\Psi \otimes \Psi' \simeq \Psi' \otimes \Psi$

Channel equality is a *partial equivalence* which means that there can be terms that are not equivalent with anything (not even themselves). This does not allow them to be used as channels (but only as data). The composition of assertions (wrt. assertion equivalence) must be associative, commutative, and have **1** as unit; moreover, composition must preserve equivalence of assertions.[2,3]

The intuition is that assertions will be used to capture assumptions about the environment of the processes. Conditions will be used as guards for guarded (non-deterministic) choices, and are to be tested against the assertion of the environment for entailment. Terms are used to represent complex data communicated through channels, but will also be used to define the channels themselves, which can thus be more than just mere names, as is the in pi-calculus. The composition of assertions should capture the notion of combining assumptions from several components of the environment.

Definition 2.2.5 (syntax): The syntax for building Psi-processes is the following (Psi-processes are denoted by $P, Q, \ldots$; terms from $\mathbf{T}$ by $M, N, \ldots$):

$\mathbf{0}$	Empty/trivial process
$\overline{M}\langle N \rangle.P$	Output
$\underline{M}\langle(\lambda\tilde{x})N\rangle.P$	Input
$\mathbf{case}\ \varphi_1 : P_1, \ldots, \varphi_n : P_n$	Conditional (non-deterministic) choice
$(\nu a)P$	Restriction of name a inside processes P
$P \,\|\, Q$	Parallel composition
$!P$	Replication
$(\!\|\Psi\|\!)$	Assertion processes

where $\tilde{x}$ is a sequence of variable names bound in the object term N, $\varphi_i \in \mathbf{C}$ are conditions, a is a name possibly appearing in P, and $\Psi \in \mathbf{A}$ is an assertion.

[2] Note that *idempotence* ($\Psi \otimes \Psi \simeq \Psi$) is not required from the composition operation, meaning that logics to represent resources, like linear logic, can be captured through the assertions language.

[3] Note also that *weakening* ($\Psi \vdash \varphi \Rightarrow \Psi \otimes \Psi' \vdash \varphi$) is not required, meaning that non-monotonic logics could be captured as well.

The *input* and *output* processes are as in pi-calculus except that the channel objects M can be arbitrary terms. In the input process the object $(\lambda\tilde{x})N$ is a pattern with the variables $\tilde{x}$ bound in N as well as in the continuation process P.[4] Intuitively, any term message received on M must match the pattern N for some substitution of the variables $\tilde{x}$. The same substitution is used to substitute these variables in P after a successful match. The traditional pi-calculus input $a(x).P$ would be modelled in Psi-calculi as $\underline{a}\langle(\lambda x)x\rangle.P$, where the names are the only terms allowed. Restriction, parallel composition, and replication are the standard constructs of pi-calculus.

The **case** process behaves like one of the P_i for which the condition φ_i is entailed by the current environment assumption, as defined by the notion of *frame* which we present later. Frames are familiar from the applied pi-calculus [AF01], where were introduced with the purpose of capturing static information about the environment (or seen in reverse, the frame is the static information that the current process exposes to the environment). Particular examples of using the case construct are:

1. **case** $\varphi : P$ which can be read as **if** φ **then** P;
2. **case** $\top : P_1, \top : P_2$, where $\top$ would be any condition that is entailed by all assertions (like $a \leftrightarrow a$ in pi-calculus); this use is mimicking the pi-calculus non-deterministic choice $P_1 + P_2$.

Remark 2.2.6: Psi-calculi work with finite terms and processes. Therefore, we restrict our further investigations to finite event structures and DCRs. To handle event structure over an infinite set of events one needs to investigate extensions of Psi-calculi with three forms of infinity:

1. Infinite summation, which is sometimes found in process algebras, e.g., in Milner's SCCS [Mil83]. In the case of Psi-calculi an infinite case construct can be written as **case** $\tilde{\varphi_i} : \tilde{P_i}$ where infinite lists are used to represent the respective condition/process pairs. No significant changes to the semantics would be needed.
2. Infinite parallel composition could use the same semantic rule as for the finite case, but care needs to be taken with the required notions of frame and entailment. Often the replication is the preferred way to obtain infinite parallel components.
3. Infinite nominal data structures, where works into infinite terms would be a starting point.

Assertion processes $(\!|\Psi|\!)$ can float freely in a process (i.e., through parallel compositions) thus describing assumptions about the environment. Otherwise, assertions can appear at the end of a sequence of input/output actions, i.e., these are the guarantees that a process provides after it makes an action (on the same lines as in assume/guarantee reasoning about programs). Assertion processes are somehow similar to the active substitutions of the applied pi-calculus, except that assertions do not have computational behaviour, but only restrict the behaviour of the other constructs by providing their assumptions about the environment.

Example 2.2.7 (Pi-calculus as an instance): To obtain pi-calculus [MPW92] as an instance of Psi-calculi use the following, built over a single set of names $\mathcal{N}$:

$$\begin{array}{lcl}
\mathbf{T} & \triangleq & \mathcal{N} \\
\mathbf{C} & \triangleq & \{a = b \mid a, b \in \mathbf{T}\} \\
\mathbf{A} & \triangleq & \{\mathbf{1}\} \\
\dot{\leftrightarrow} & \triangleq & = \\
\vdash & \triangleq & \{(\mathbf{1}, a = a) \mid a \in \mathbf{T}\}
\end{array}$$

with the trivial definition for the composition operation. The only terms are the channel names $a \in \mathcal{N}$, and there is no other assertion than the unit. The conditions are equality tests for channel names, where the only successful tests are those where the names are equal. Hence, channel comparison is defined as just name equality.

[4] Note the use of λ as a syntactic binder denoting patterns of terms, and the use of the standard pi-calculus restriction operation on names ν. The use of λ is only in the input terms.

Example 2.2.8: From the instance created in Example 2.2.7 one can obtain the polyadic pi-calculus [CM03] by adding tupling symbols t_n for tuples of arity n to $\mathbf{T}$, i.e.

$$\mathbf{T} = \mathcal{N} \cup \{t_n(M_1, ..., M_n) : M_1, ...M_n \in \mathbf{T}\}.$$

The polyadic output is to simply output the corresponding tuple of object names, and the polyadic input $a(b_1, ..., b_n).P$ is represented by a pattern matching

$$\underline{a}(\lambda b_1, ...b_n)t_n(b_1, ..., b_n).P.$$

Strictly speaking this allows nested tuples as well as tuples in the subject position of inputs and outputs. But these do not give rise to transitions because the definition of channel equality only applies to channel names, thus $M \dot{\leftrightarrow} M$ can be entailed by an assertion only when M is a name.

Psi-calculi are given an operational semantics in [BJPV11] using labelled transition systems, where the states are the process terms and the transitions represent one reduction step, labelled with the action that the process executes. The actions, generally denoted by α, β, represent respectively the input and output constructions, as well as τ the internal synchronisation/communication action:

$$\overline{M}\langle(\nu\tilde{a})N\rangle \mid \underline{M}\langle N\rangle \mid \tau$$

The restriction operator ν binds the names $\tilde{a}$ in N. We will denote by $bn(\alpha)$ the set of bound names in a communication term; i.e., $bn(\overline{M}\langle(\nu\tilde{a})N\rangle) = \tilde{a}$.

Transitions are done in a context, which is represented as an assertion Ψ, capturing assumptions about the environment:

$$\Psi \rhd P \xrightarrow{\alpha} P'$$

Intuitively, the above transition could be read as: The process P can perform an action α in an environment respecting the assumptions in Ψ, after which it would behave like the process P'.

The environment assertion is obtained using the notion of *frame* which essentially collects (using the composition operation) the outer-most assertions of a process. A frame also keeps the information about the restrictions under which the assertion processes are found.

Definition 2.2.9 (frame): A *frame* is of the form $(\nu\tilde{b})\Psi$ where $\tilde{b}$ is a sequence of names that bind into the assertion Ψ. We write just Ψ for $(\nu\epsilon)\Psi$ when there is no risk of confusing a frame with an assertion. We identify alpha variants of frames. In consequence, composition of frames is defined by $(\nu\tilde{b_1})\Psi_1 \otimes (\nu\tilde{b_2})\Psi_2 = (\nu\tilde{b_1}\tilde{b_2})\Psi_1 \otimes \Psi_2$ where $\tilde{b_1} \notin n(\tilde{b_2}, \Psi_2)$ and vice versa. The frame of a process $\mathcal{F}(P)$ is defined inductively on the structure of the process as:

$$\begin{aligned}
&\mathcal{F}(\llparenthesis\Psi\rrparenthesis) = \Psi\\
&\mathcal{F}(P \,||\, Q) = \mathcal{F}(P) \otimes \mathcal{F}(Q)\\
&\mathcal{F}((\nu a)P) = (\nu a)\mathcal{F}(P)\\
&\mathcal{F}(!P) = \mathcal{F}(\mathbf{case}\ \tilde{\varphi} : \tilde{P}\,) = \mathcal{F}(\overline{M}\langle N\rangle.P) = \mathcal{F}(\underline{M}\langle(\lambda\tilde{x})N\rangle.P) = \mathbf{1}
\end{aligned}$$

Any assertion that occurs under an action prefix or a condition is not visible in the frame.

Example 2.2.10: Calculating the frame of the following process, when $a \notin n(\Psi_1)$, is:

$$\mathcal{F}(\llparenthesis\Psi_1\rrparenthesis \,||\, (\nu a)(\llparenthesis\Psi_2\rrparenthesis \,||\, \overline{M}\langle N\rangle.\llparenthesis\Psi_3\rrparenthesis)) = \Psi_1 \otimes (\nu a)\Psi_2 = (\nu a)(\Psi_1 \otimes \Psi_2).$$

Here Ψ_3 occurs under a prefix and is therefore not included in the frame. An agent where all assertions are guarded thus has a frame equivalent to $\mathbf{1}$. Because frames are considered equivalent up to alpha-conversion, proper renaming allows to move restriction operators (νa), as exemplified here.

Definition 2.2.11 (semantics): The transition rules for Psi-calculi are the following, where the symmetric rules for (PAR) and (COM) are elided.

$$\frac{\Psi \vdash M \leftrightarrow K}{\Psi \rhd \underline{M}\langle(\lambda\tilde{y})N\rangle.P \xrightarrow{\underline{K}N[\tilde{y}:=\tilde{L}]} P[\tilde{y} := \tilde{L}]} \;(\textsc{in})$$

$$\frac{\Psi \vdash M \leftrightarrow K}{\Psi \rhd \overline{M}\langle N\rangle.P \xrightarrow{\overline{K}N} P} \;(\textsc{out})$$

$$\frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \varphi_1 : P_1, \ldots, \varphi_n : P_n \xrightarrow{\alpha} P'} \;(\textsc{case})$$

$$\frac{\Psi_Q \otimes \Psi_P \otimes \Psi \vdash M \leftrightarrow K \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{K}N} Q' \qquad \Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{M}(\nu\tilde{a})N} P' \quad \tilde{a} \notin n(Q)}{\Psi \rhd P \,||\, Q \xrightarrow{\tau} (\nu\tilde{a})(P' \,||\, Q')} \;(\textsc{com})$$

In the (COM) rule the assertions Ψ_P and Ψ_Q come from the frames of $\mathcal{F}(P) = (\nu\tilde{b}_P)\Psi_P$ respectively $\mathcal{F}(Q) = (\nu\tilde{b}_Q)\Psi_Q$ and it is assumed that $\tilde{b}_P$ is fresh for all of $\Psi, \tilde{b}_Q, Q, M$ and P, and respectively for $\tilde{b}_Q$.

$$\frac{\Psi \otimes \Psi_Q \rhd P \xrightarrow{\alpha} P' \qquad bn(\alpha) \notin n(Q)}{\Psi \rhd P \,||\, Q \xrightarrow{\alpha} P' \,||\, Q} \;(\textsc{par})$$

$$\frac{\Psi \rhd P \,||\, !P \xrightarrow{\alpha} P'}{\Psi \rhd !P \xrightarrow{\alpha} P'} \;(\textsc{rep})$$

$$\frac{\Psi \rhd P \xrightarrow{\alpha} P' \qquad b \notin n(\alpha), b \notin n(\Psi)}{\Psi \rhd (\nu b)P \xrightarrow{\alpha} (\nu b)P'} \;(\textsc{scope})$$

$$\frac{\Psi \rhd P \xrightarrow{\overline{M}(\nu\tilde{a})N} P' \qquad b \notin n(\tilde{a}), b \notin n(\Psi), b \notin n(M) \qquad b \in n(N)}{\Psi \rhd (\nu b)P \xrightarrow{\overline{M}(\nu\tilde{a}\cup\{b\})N} P'} \;(\textsc{open})$$

There is no transition rule for the assertion process; this is only used in constructing frames. Once an assertion process is reached, the computation stops, and this assertion remains floating among the other parallel processes and will be composed part of the frames, when necessary, like in the case of the communication rule. The empty process has the same behaviour as, and thus can be modelled by, the trivial assertion process $(\!|\mathbf{1}|\!)$.

The (IN) rule makes transitions labelled with any channel term K equivalent to the input channel M, and for any substitution replacing the variables $\tilde{y}$ by term values $\tilde{L}$ in the (pattern) term N. The input rule is open to any possible matching outputs, where the (COM) rule will pair any of the exact matchings. The (OUT) rule just outputs the term N on some equivalent channel term K.

The (CASE) rule shows how the conditions are tested against the context assertions. From all the entailed conditions one is non-deterministically chosen as the continuation branch.

The communication rule (COM) shows how the environment processes executing in parallel contribute their top-most assertions to make the new context assertion for the input/output action of the other parallel process. The (COM) rule requires that for a synchronization to happen the channels in the transition labels for the input and output processes must be equivalent.

The (PAR) rule allows a component P in a parallel process to do an α transition to P' as long as the bound names of the transition label are not captured by the environment process Q, i.e., when the bound names of α are fresh in Q ($bn(\alpha) \cap n(Q) = \emptyset$). Moreover, for the frame $\mathcal{F}(Q) = (\nu\tilde{b}_Q)\Psi_Q$ it is assumed that $\tilde{b}_Q$ is fresh for Ψ, P and α.

The (REP) rule is standard from pi-calculi.

The (SCOPE) rule can be applied only when b is fresh in both α and the assertion that the process is executed with.

In (OPEN) the expression $\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

Example 2.2.12: For a simple example of a transition, suppose for an assertion Ψ and a condition φ that $\Psi \vdash \varphi$. Also assume that

$$\forall \Psi'.\Psi' \rhd Q \xrightarrow{\alpha} Q'$$

i.e., Q has an action α regardless of the environment. Then by the (CASE) rule we get

$$\Psi \rhd \textbf{case } \varphi : Q \xrightarrow{\alpha} Q'$$

i.e., **case** $\varphi : Q$ has the same transition if the environment is Ψ. Since $\mathcal{F}((\!|\Psi|\!)) = \Psi$ and $\Psi \otimes \mathbf{1}$ we get by (PAR) that

$$\mathbf{1} \rhd (\!|\Psi|\!) \,||\, \textbf{case } \varphi : Q \xrightarrow{\alpha} (\!|\Psi|\!) \,||\, Q'.$$

2.2.1 Terms

A more detailed introduction to nominal sets used in Psi-calculi can be found in [BJPV11, Sec.2.1] and the recent book [Pit13] contains a thorough treatment of both the theory behind nominal sets as well as various applications, e.g., see [Pit13, Ch.8] for nominal algebraic datatypes. For our presentation here we expect only some familiarity with notions of algebraic datatypes and term algebras. In the following we briefly present the notion of terms that we will be using in our encodings in the rest of the chapter.

Definition 2.2.13 (terms, cf. [BN98, Chap.3.1]): In universal algebra, *terms* are constructed from a signature $\mathcal{F}$ of function names of some arity, and a set of variables $\mathcal{X}$, and are denoted $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Function symbols of arity 0 are called constants. Function symbols are sometimes denoted as $f(_)$ or $f(_,_)$ to emphasise their arity (i.e., number of arguments, respectively one and two in this example). Terms without variables are called *ground*, their set being denoted by $\mathcal{T}(\mathcal{F})$, whereas terms containing variables are sometimes called *open*, to emphasise this aspect (in consequence, some works call ground terms *closed*). Every variable is also a term, i.e., an open term.

One could see an intuitive association between variables and names, since in nominal datatypes names can be subject to substitutions, the same as variables in open terms. Though names have other properties, as we have seen, like bindings or alpha conversion.

Example 2.2.14 (natural numbers as ground terms): The set of natural numbers can be seen as terms denoted by $\mathbb{N}$ and defined as $\mathcal{T}(\{s(_), 0\})$, i.e., only ground terms, built from the single constant term 0 using the unary successor function. An example of the term representing number 3 is $s(s(s(0)))$.

Definition 2.2.15 (multi-sorted terms, cf. [GM92]): A multi-sorted algebraic structure is obtained if we add a notion of *sorts* to the Definition 2.2.13. Consider a set of sorts $s \in \mathcal{S}$ with a partial order on them, e.g., $s_1 < s_2$ means that any term of sort s_1 is also of sort s_2. Assign to each function symbol $f \in \mathcal{F}$ a sort for each parameter and a sort for output; e.g., $f : s_1, \ldots, s_n \to s$ takes n arguments of the respective sorts. In particular, each constant symbol is of some sort. One function can now be applied only to terms of the appropriate sort, to produce a term of the respective result sort. The set of variables is now partitioned into sorted variables, i.e., each variable is of a certain sort. We sometimes mention the set of sorts as $\mathcal{T}(\mathcal{F}, \mathcal{X}, \mathcal{S})$ when we want to be specific; but most of the time we rely on the context for disambiguation.

Definition 2.2.16 (equality of terms): *Equations* can be defined between terms, $t_l = t_r$, to express which terms should be viewed as equal. Equations are usually defined between open terms and are closed under substitutions, meaning that any two terms obtained by applying the same substitution to both t and t' would also be considered equal. For some set of equations, we infer the equality of two terms by applying the inference rules of equational logic, i.e., identity, symmetry, transitivity, closure under function application and under substitutions.

Example 2.2.17 (multi-sets as ground multi-sorted terms): One representation of sets can be given as terms built over two sorts $\mathcal{S} = \{el, set\}$ using constant symbols e_i of sort el and constant symbol $\emptyset$ of sort set, and using a multi-sorted concatenation operation $_{el} : _{set}$ which takes the first argument of sort el and the second argument of sort set. An example is $e_1 : (e_2 : (e_3 : \emptyset))$. Because elements can appear several times in a concatenation, we

have just modelled *multi-sets*. All multi-sets over some $e_i \in E$ are denoted by $\mathbb{N}^E$. With standard equations one could treat the multi-set terms with duplicate elements as equal to terms with a single copy of the element so that, e.g., $e_1 : (e_1 : \emptyset)$ would be equal to $e_1 : \emptyset$. Examples of equations of relevance to this case are: commutativity in multi-sets could be $e_1 : (e_2 : x) = e_2 : (e_1 : x)$, whereas an annihilation equation would be $e_1 : (e_1 : x) = e_1 : x$. We denote all the terms capturing sets over some E of constant symbols by 2^E.[5]

Example 2.2.18 (natural numbers with operations): When interested in operations on natural numbers we could build a term algebra $\mathbb{N}_{op}$ from $\mathbb{N}$ of Example 2.2.14 by adding operators and equations related to the operators, as well as variables. Consider two sorts $\mathcal{S} = \{g, op\}$ and have all ground terms from Example 2.2.14 to be of sort g, i.e., put the constant 0 and the successor function to be of sort g. Take the order $g < op$, saying that any ground number g is also a number term with operations. Now define any operations, like $_+_$ or $_-_$, to be of sort op, i.e., taking as input op terms and returning op terms. Put the standard definition of such an operator into equations, e.g.: $s(0)+x = s(x)$. One can also put as equations the standard properties of such operators, like commutativity. So in our example $\mathbb{N}_{op} = (\{s, 0, +, -\}, \mathcal{X}, \mathcal{S})$. The sorts have been used just to make the representation nicer, i.e., having the natural numbers as building blocks on which the operations work. But in the presence of equations we can safely do without sorts; we will just have the successor function possibly applied to an operation like $s(s(0) + s(s(0)))$.

With such a definition of natural numbers with operations as terms, we can then work with them as usual in mathematical proofs, but also as terms when the Psi-calculi rigour requires it. In particular, when using a proof assistant, as is customarily done when making meta-proofs for Psi-calculi, we would need to select an appropriate package to work with natural numbers, and with sets and multi-sets. These packages would be using encodings on the lines described above, to every detail. For this chapter we stick to the well-known intuitive notations for natural numbers and multi-sets.

Example 2.2.19 (multi-sets with operations): Take the sets and multi-sets of Example 2.2.17 and add functional symbols for standard operations like: $_\cup_$, $_\setminus_$, $_+_$ (the last one standing for summation of multi-sets). Consider the definitions of these operations in the equations; for example $e_1 : e_2 : \emptyset \cup e_3 : \emptyset = e_1 : e_2 : e_3 : \emptyset$. Variables $\mathcal{X}$ are included as well. We could also add sorts for ground sets and sets with operators, as we did in the previous example. We denote such sets and multi-sets with operators and variables over some E by $2^E_{op} = (\{:, \emptyset, \cup, \setminus\}, \mathcal{X})$ and $\mathbb{N}^E_{op} = (\{:, \emptyset, +, -\}, \mathcal{X})$.

Many times data structures used in computer science are multi-sorted, and thinking in terms of sorts makes our results easier to follow. Therefore, we give a few definitions for sorts in the case when names are present. Complete treatment can be found in references like [Pit13, UPG04, BGP+15, BGP+14].

Definition 2.2.20 (multi-sorted nominal datatypes, cf. [Pit13, ch.8] or [UPG04]): Consider a set of *name sorts* $\mathcal{S}^{\mathcal{N}}$ disjoint from the set of sorts used for the datastructures, which we will call *data sorts* and denote $\mathcal{S}^{\mathcal{D}}$. Each name is assigned a name sort, the same as we were doing for variables. Name swapping is now *sort-respecting* in the sense that the two names being swapped must have the same name sort. In consequence, *freshness* and *name abstraction* are also sort-respecting (see [Pit13, ch.4.7]). Nominal datastructures are built over sorts described using the following grammar:

$$\mathcal{S} ::= \mathcal{S}^{\mathcal{N}} \mid \mathcal{S}^{\mathcal{D}} \mid 1 \mid \mathcal{S}^{\mathcal{N}}\!: \mathcal{S} \mid (\mathcal{S}, \mathcal{S})$$

This basically describes binding sorts and pairs (and thus tuples) sorts. Functions are defined as $f : \mathcal{S} \to \mathcal{S}^{\mathcal{D}}$ always returning a data sort. Terms are built respecting the sorting, including information about name binding.

Our use of sorts in Psi-calculi is rather simplistic, mainly to make sure that the right kind of terms are being used in the right place; e.g., when receiving data on a channel. We prefer to minimise mentioning sorting aspects, as for our results these details would mean too much cluttering without gained insights or correctness concerns. Nevertheless, when strictness is necessary, like when working with a proof assistant, then all details of the sorting should be in place, and the methods described in [BGP+15, BGP+14] should be followed. We give here a brief definition of some main aspects of sorted Psi-calculi.

[5] Note that to model infinite sets we would need infinite terms in the above term encoding.

Definition 2.2.21 (sorted Psi-calculi, [BGP+15, sec.2.4]): Multi-sorted Psi-calculi use two main notions on top of multi-sorted nominal datatypes:

sorts for channels specify for each sort (designating terms that can be the subject of a channel) the sort of terms (objects) that can be send/received on that channel;

multi-sorted substitutions need to know which sorts (of terms) can substitute which sorts of names; i.e., a relation $<_{sub} \subseteq \mathcal{S}_{\mathcal{N}} \times \mathcal{S}$. The relation on sorts from Definition 2.2.15 is respected in the sense that if $a <_{sub} s_2$ and $s_1 < s_2$ then $a <_{sub} s_1$.

In our work we rely on sorts to give some discipline in building Psi-terms without cluttering unnecessarily the notation. Sorts are intuitive and we will abuse the notation and rely on the context and intuition to disambiguate. Here are a few examples of our simple way of using sorts in the Psi-instances that we will define.

Example 2.2.22 (simple use of sorts in Psi-instances): Often a function like pairing is multi-sorted, $(_,_) : s_1 \times s_2 \to s_3$. The left parameter may be of sort natural numbers whereas the right parameter may be a multiset; everything could be considered of sort *pairs*. We would like to allow names to be used as parameters, and these names could be replaced upon a Psi-communication. In this case we just say that the names must be of sort natural number and multiset. In consequence, the substitutions to respect the sorts should replace the name on the left only with natural number terms and the name on the right only with multisets.

2.3 Representing event structures in Psi-calculi

In this section we provide a Psi-calculus representation of finite *prime* event structures (recalled in Definition 2.3.1).

It is fairly easy to represent the interleaving, transition semantics for a finite event structure as a Psi-calculus term. However, in contrast to most process calculi, event structures and more expressive event-based models of concurrency [Win82, Pra91, Pra00, vG06, vGP09, Joh15, Gup94, Pra95] come with a non-interleaving semantics. A non-interleaving semantics makes it possible to distinguish between interleaving and independence (sometimes called “true” concurrency) and are well behaved wrt. action refinement [vGG01]. A simple example is given by two concurrent processes executing each a different instance of the same action a. An interleaving transition system based model would represent such a process by an ”interleaving” diamond with all four sides labelled by the same action, which semantically typically would be equal to the sequential composition $a.a$ of the two actions. Refining the action a into $a1.a2$ in the semantical model, would thus result in the single sequence $a1.a2.a1.a2$ as the possible behaviour. However, when refining the parallel composition of two concurrent processes that both executes a, one would expect all possible interleavings, that is, the two different behaviours $\{a1.a2.a1.a2, a1.a1.a2.a2\}$.

The encoding of finite prime event structures into the instance of Psi-calculi, which we call eventPsi, not only preserves the behaviour of event structures up to interleaving diamonds, but it also preserves the causal structure by exploiting the assertions and conditions of Psi-calculi, and as a consequence is also compatible with action refinement.

We show in the subsequent section that this idea can be generalized to DCR-graphs, and we believe that also other generalized versions of event structures [Win87, vGP09] can be represented as Psi-calculi following a similar approach as presented here.

2.3.1 Background on event structures

We follow the standard notation and terminology from [WN95, sec.8].

Definition 2.3.1 (prime event structures): *A labelled prime event structure* over alphabet Act is a tuple $\mathcal{E} = (E, \leq, \sharp, l)$ where E is a possibly infinite set of events, $\leq \subseteq E \times E$ is a partial order (the *causality* relation) satisfying

1. *the principle of finite causes*, i.e.: $\forall e \in E : \{d \in E \mid d \leq e\}$ is finite,

and $\sharp \subseteq E \times E$ is an irreflexive, symmetric binary relation (the *conflict* relation) satisfying

2. *the principle of conflict heredity*, i.e., $\forall d, e, f \in E : d \leq e \wedge d \sharp f \Rightarrow e \sharp f$.

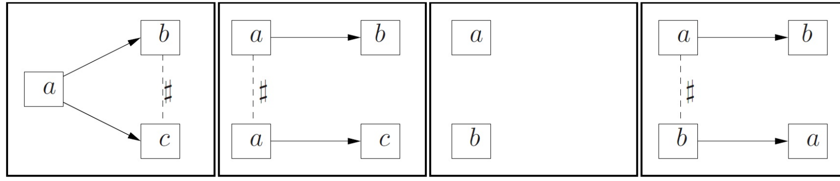


Fig. 2.2: Classic examples of event structures. Different boxes represent different events, possibly labelled the same. Arrows represent causality (not displaying those coming from the transitive closure); and dashed lines represent the symmetric conflicts (not displaying those coming from heredity).

and $l : E \to Act$ is the labelling function. In the rest of the paper we only consider finite prime event structures, that is prime event strctures where the set of events E is finite. Denote by $\mathbb{E}$ the class of all finite prime event structures.

Intuitively, a prime event structure models a concurrent system by taking $d \leq e$ to mean that event d is a prerequisite of event e, i.e., event e cannot happen before event d has been done. A conflict $d\sharp e$ says that events d and e cannot both happen in the same run. Compared to other models of concurrency like process algebras, event structures model systems by looking only at their events, and how these events relate to each other. The two basic relations considered by event structures are the *dependency* and the *conflict* relations. The conflict relation can be used to capture choices made by the system, since the execution of one event discards all other events in conflict with itself for the rest of the computation.

Labels can be understood as actions, with a wide and general meaning. Events are instances of actions, and an action can happen several times, thus as different events. The same action can also happen in different components running in parallel, giving rise to *autoconcurrency*, as exemplified in the beginning of this section. Actions are important for observational equivalence, but not only them (see Example 2.3.5).

Example 2.3.2: In Figure 2.2 we pictured four simple examples of finite event structures (taken from [vGV97, Fig.4]). We illustrate events as boxes containing their labels, the dependency relation by arrows, and the conflict relation by dashed lines with a $\sharp$ sign. In the left-most event structure we have three events, where the events labelled b and c depend on the event labelled a and are in conflict with each other. This is a standard branching point which could be specified in a simple CCS notation as $a; (b + c)$. In the second event structure we have two (conflicting) events, both labelled with a, and two events labelled b and c which depend on the first and the second a-labelled event respectively. Because of the principle of conflict hereditary, b is in conflict with the lower a-labelled event and similarly, c is in conflict with the upper a-labelled event (but the conflict relations are in this case usually not explicitly illustrated). In CCS notation this could be $a; b + a; c$. In the third event structure we have just two events without any explicit or inherited relation, which means that they are *concurrent* (e.g., $a||b$), as made precise below. The last event structure is similar to the second, except that it offers two conflicting paths with the label a followed by b or b followed by a respectively.

Definition 2.3.3 (concurrency): *Causal independence (concurrency)* between events is defined in terms of the above two relations as

$$d||e \triangleq \neg(d \leq e \vee e \leq d \vee d\sharp e).$$

This definition captures the intuition that two events are concurrent when there is no causal dependence between the two and, moreover, they are not in conflict.

From the definition it follows that only the two events in the third event structure in Figure 2.2 are concurrent.

The behaviour of an event structure is described by subsets of events that happened in some (partial) run of the system being modelled. This is called a *configuration* of the event structure, and *steps* can be defined between configurations.

Definition 2.3.4 (configurations): Define a *configuration* of an event structure $\mathcal{E} = (E, \leq, \sharp)$ to be a finite subset of events $C \subseteq E$ that respects:

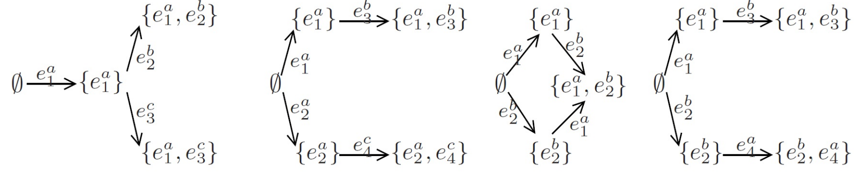


Fig. 2.3: Configurations and steps for event structures in Figure 2.2. The labels of the events are shown as superscripts.

1. *conflict-freeness*: $\forall e, e' \in C : \neg(e \sharp e')$ and,
2. *downwards-closure*: $\forall e, e' \in E : e' \leq e \wedge e \in C \Rightarrow e' \in C$.

We denote the set of all configurations of some event structure by $\mathbb{C}_{\mathcal{E}}$.

Conflict-freeness is saying that no two conflicting events can happen in one run. This also says that once an event is discarded it can never be executed on the current run. This is similar to how the semantics of the choice operator in process algebras is defined (see rule (CASE) in Section 2.2) where all other branches of the choice are discarded once a step is taken. The downwards-closure says that all the dependencies of an executed event (i.e., which is part of a configuration) must have been executed also (on this same run).

Note in particular that $\emptyset$ is a configuration (i.e., the root configuration) and that any set $\lceil e \rceil \triangleq \{e' \in E \mid e' \leq e\}$ is also a configuration determined by the single event e. Events determine steps between configurations in the sense that $C \xrightarrow{e} C'$ whenever C, C' are configurations, $e \notin C$, and $C' = C \cup \{e\}$.

Example 2.3.5: For the examples from Figure 2.2 we get the configurations and steps depicted in Figure 2.3.

One may note that if only the paths of labels are observed, the two first event structures are indistinguishable, but if the branching structure is observed, i.e. by a bisimulation equivalence, they are distinguishable. One may also note that if the paths of labels are observed and even if branching time is observed, the two latter event structures are indistinguishable, but if concurrency is observed, i.e. by a history-preserving bisimulation equivalence [DDM88, vGG01], they are distinguishable.

Remark 2.3.6: It is known (see e.g., [WN95, Prop.18]) that prime event structures are fully determined by their sets of configurations, i.e., the relations of causality, conflict, and concurrency can be recovered only from the set of configurations $\mathbb{C}_{\mathcal{E}}$ as follows:

1. $e \leq e'$ iff $\forall C \in \mathbb{C}_{\mathcal{E}} : e' \in C \Rightarrow e \in C$;
2. $e \sharp e'$ iff $\forall C \in \mathbb{C}_{\mathcal{E}} : \neg(e \in C \wedge e' \in C)$;
3. $e || e'$ iff $\exists C, C' \in \mathbb{C}_{\mathcal{E}} : e \in C \wedge e' \notin C \wedge e' \in C' \wedge e \notin C' \wedge C \cup C' \in \mathbb{C}_{\mathcal{E}}$.

It is also known (see e.g., [WN95, Sec.8] for prime event structures or [vGG01, Sec.4] for the more general event structures of [Win87]) that there is no loss of expressiveness when working with finite, instead of infinite configurations. An infinite configuration can be obtained from infinite union of finite configurations coming from an infinite run.

For some event e we denote by $\leq e = \{e' \in E \mid e' \leq e\}$ the set of all events which are conditions of e (which is the same as the notation $\lceil e \rceil$ from [WN95], but we prefer to use the above so to be more consistent with similar notations we use in the rest of this paper for similar sets defined for DCRs too), and $\sharp e = \{e' \in E \mid e' \sharp e\}$ those events in conflict with e. We denote by $< e = \leq e \setminus \{e\}$ the non-trivial conditions of e, i.e., excluding itself.

2.3.2 The encoding of prime event structures

In this section we provide an encoding of finite prime event structures into an instance of Psi-calculi which we call eventPsi. We consider *finite* event structures only, since it is not the goal of our work to represent denotational models. Moreover, the direct encoding we are aiming for of event structures with infinite sets of events would require a treatment in Psi-calculi of infinite parallel composition, infinite terms, frame definition, and careful look at the SOS rules which use entailment among infinite assertion and condition terms. Such a treatment is beyond the scope of the paper. We will instead see in the next section how to encode an event-based model generalising event structures to allow finite representations of infinite behaviour.

The intuition of the encoding is to represent event structure configurations as assertions and the causality and conflict relations as conditions. We do this by taking assertions $\mathbf{A}$ to be sets of events representing the history of executed events, i.e. a configuration, and composition just set union. Conditions $\mathbf{C}$ are taken to be pairs (p, c) of sets of events and entailment relation is then defined such that p represents the preconditions of an event, i.e. $\leq e$, and c represents the conflict set, i.e. $\sharp e$. That is, an assertion (history of executed events) Ψ entails a condition (p, c) if and only if $p \subseteq \Psi$ (the preconditions have been executed) and $c \cap \Psi = \emptyset$ (none of the conflicting events have been executed). Finally, we take the set of terms $\mathbf{T}$ to be simply a set of constants representing the events. To keep the exposition simple, we will ignore event labels until Section 2.3.3, where we treat action refinement. But labels could easily be added by redefining the terms to be pairs of an event and its label, for some given labelling function; e.g., $(e, l(e))$.

Definition 2.3.7 (event Psi-calculus over E): We define a Psi-calculus instance, called *eventPsi*, parametrised by a set E of constant symbols, to be understood as *events*, by providing the following definitions of the key elements of a Psi-calculus instance:

$$\mathbf{T} \stackrel{def}{=} E$$

$$\mathbf{C} \stackrel{def}{=} (2^E \times 2^E) \cup \{e \leftrightarrow f \mid e, f \in \mathbf{T}\}$$

$$\mathbf{A} \stackrel{def}{=} 2^E$$

$$\otimes \stackrel{def}{=} \cup$$

$$\mathbf{1} \stackrel{def}{=} \emptyset$$

$$\vdash \stackrel{def}{=} \begin{cases} \Psi \vdash (D, C) \text{ iff } (D \subseteq \Psi) \wedge (C \cap \Psi = \emptyset) \\ \Psi \vdash e \leftrightarrow f \text{ iff } e = f \end{cases}$$

where $\mathbf{T}$, $\mathbf{C}$, and $\mathbf{A}$ are algebraic data types built over the constants in E.

It is easy to see that our definitions respect the restrictions of making a Psi-calculus instance. In particular, channel equivalence is symmetric and transitive since equality is. The $\otimes$ is compositional, associative and commutative, as $\cup$ is; and moreover $\emptyset \cup S = S$, for any set S, i.e., $\mathbf{1}$ is the identity for $\otimes$.

Remark 2.3.8: We are not using the nominal aspects of Psi-calculi. Throughout the rest of this section we do not work with names, and therefore the support of all terms will be empty. Names will make their appearance in the encoding of DCRs in Section 2.4.

We are now ready to provide the encoding ESPSI which maps a finite prime event structure and a configuration to an eventPsi-process. The eventPsi-process is defined as a parallel composition of atomic "event processes". These come in two forms: The first, defined simply as an assertion process, corresponds to events in the configuration of the translated event structure (i.e., those that already happened). The latter corresponds to events that have not happened yet and are defined using the case construct with a condition $\varphi_e = (p, c)$ where $p = <e$, i.e. the preconditions of the event e, and $c = \sharp e$ is the set of events that e is in conflict with. The definition of entailment then ensures that the **case** process can execute if and only if the event is enabled, and if it executes, the event is asserted, thereby updating the configuration. To easily observe which event happened, we also communicate the event e on the channel e.

Definition 2.3.9 (event structures to eventPsi): For $\mathcal{E} = (E, \leq, \sharp)$ an event structure and a configuration C of $\mathcal{E}$, define $\textsc{espsi}(\mathcal{E}, C)$ as

$$\textsc{espsi}(\mathcal{E}, C) \;=\; ||_{e \in E} P_e$$

with

$$P_e = \begin{cases} (\!|\{e\}|\!) & \text{if } e \in C \\ \mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!) & \text{otherwise} \end{cases}$$

where $\varphi_e = (<e, \sharp e)$. If the configuration is empty we will allow writing $\textsc{espsi}(\mathcal{E})$ for $\textsc{espsi}(\mathcal{E}, \emptyset)$.

We often use the product notation in situations as above, i.e., $\prod_{e \in E} P_e$ to mean the parallel composition of the P_e processes.

We have seen that the eventPsi-processes that we obtain from event structures in Definition 2.3.9 have a specific syntactic form. But the eventPsi instance allows any process term to be constructed over the three nominal data-types that we gave in Definition 2.3.7. Below we give the syntactic restrictions on eventPsi-process terms corresponding to event structures, via the mapping defined by Theorem 2.3.19.

Definition 2.3.10 (syntactic restrictions for eventPsi): We define an eventPsi-process to be *syntactically correct* if it is constructed using the grammar:

$$P_{ES} \quad := \quad (\!|\{e\}|\!) \quad | \quad \mathbf{case}\ \varphi : \overline{e}\langle e\rangle.(\!|\{e\}|\!) \quad | \quad P_{ES} \,||\, P_{ES}$$

and moreover, it respects the following constraints, for any $\varphi_e, \varphi_{e'}$ from $\mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)$ respectively $\mathbf{case}\ \varphi_{e'} : \overline{e'}\langle e'\rangle.(\!|\{e'\}|\!)$:

1. conflict: (i) $e \notin \pi_R(\varphi_e)$ and (ii) $e' \in \pi_R(\varphi_e) \Leftrightarrow e \in \pi_R(\varphi_{e'})$;
2. causality: (i) $e \notin \pi_L(\varphi_e)$ and (ii) $e \in \pi_L(\varphi_{e'}) \Rightarrow (e' \notin \pi_L(\varphi_e) \wedge \pi_L(\varphi_e) \subset \pi_L(\varphi_{e'}))$;
3. executed events: for any e, P_{ES} will have at most one of $(\!|\{e\}|\!)$ or $\mathbf{case}\ \varphi : \overline{e}\langle e\rangle.(\!|\{e\}|\!)$.

Denote by $ev(P) \subseteq \mathbf{T}$ the event constants appearing in a process P_{ES}.

To justify for the last restriction assume having $(\!|\{e\}|\!) \,||\, \mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)$ part of P_{ES}. This would say that e has already happened and at the same time e can happen in future when the case condition holds. This cannot be in event structures, and thus needs to be ruled out.

Definition 2.3.11 (event transitions): We define transitions between syntactically correct eventPsi processes P and P' to be $P \overset{e}{\rightsquigarrow} P'$ iff $\mathbf{1} \rhd P \xrightarrow{\overline{e}e} P'$.

Remark 2.3.12: Arbitrary eventPsi-processes can have different kinds of labelled transitions, but for syntactically correct processes the restrictions guarantee that only event transitions exist.

Lemma 2.3.13: For a syntactically correct eventPsi process P and a transition $P \overset{e}{\rightsquigarrow} P'$ then P' is also syntactically correct.

Proof: Having $P \overset{e}{\rightsquigarrow} P'$, we know from the transition rules of Psi-calculi, and the syntactic restrictions of Definition 2.3.10 that $P = \mathbf{case}\ \varphi : \overline{e}\langle e\rangle.(\!|\{e\}|\!) \,||\, Q$ where Q is syntactically correct and does not contain another (case or assertion process) P'_e, i.e., indexed by the same e. Recall that $\emptyset$ is the unit assertion $\mathbf{1}$, and that the minimal process $\mathbf{0}$ is equivalent with the $(\!|\emptyset|\!)$, which can be in place of Q so that the parallel composition ends. The transition thus is

$$\mathbf{case}\ \varphi : \overline{e}\langle e\rangle.(\!|\{e\}|\!) \,||\, Q \;\overset{e}{\rightsquigarrow}\; (\!|\{e\}|\!) \,||\, Q \;=\; P' \,.$$

As we know that Q is syntactically correct and it does not contain another P'_e then $(\!|\{e\}|\!) \,||\, Q = P'$ is also syntactically correct. ☐

Lemma 2.3.14 (correspondence configuration–frame): For any event structure $\mathcal{E}$ and configuration C, the frame of the eventPsi-process $\text{ESPSI}(\mathcal{E}, C)$ is the same as the configuration C.

Proof: Denote $\text{ESPSI}(\mathcal{E}, C) = P_E^C$ defined as in Definition 2.3.9. The frame of P_E^C is the composition with $\otimes$ of the frames of P_e for $e \in E$. As P_e is either $(\!|\{e\}|\!)$ if $e \in C$ or $\mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)$ then the frame of P_e would be either $\mathcal{F}((\!|\{e\}|\!)) = \{e\}$ or $\mathcal{F}(\mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)) = \mathbf{1} = \emptyset$. Thus the frame of P_E is the union of $\emptyset$ and all events in C. $\square$

Lemma 2.3.15 (transitions are preserved): For any event structure $\mathcal{E}$ and any of its configurations C, any transition from this configuration $C \xrightarrow{e} C'$ is matched by a transition $\text{ESPSI}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} \text{ESPSI}(\mathcal{E}, C')$ in the corresponding eventPsi-process.

Proof: By Lemma 2.3.14 the frame of $\text{ESPSI}(\mathcal{E}, C)$ is the same as C. The assumption of the lemma, i.e., the existence of the step between configurations, implies that e is enabled by the configuration C. This means that $e \notin C$, which implies by Definition 2.3.9 that $\text{ESPSI}(\mathcal{E}, C) = \mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!) \,||\, Q$. This implies that $\mathcal{F}(\text{ESPSI}(\mathcal{E}, C)) = \mathbf{1} \otimes \mathcal{F}(Q) = C$, with $e \notin C$, meaning that $\mathcal{F}(Q) = C$. Moreover, since C enables e it means that all $<e$ are in C and no $\sharp e$ is in C, which is the definition of entailment relation in eventPsi, i.e., $\mathcal{F}(\text{ESPSI}(\mathcal{E}, C)) \vdash \varphi_e$, which enables the step from $\text{ESPSI}(\mathcal{E}, C)$ that the lemma expects. After $\text{ESPSI}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} P'$ we have $P' = (\!|\{e\}|\!) \,||\, Q$ and $\mathcal{F}(P') = \mathcal{F}((\!|\{e\}|\!)) \otimes \mathcal{F}(Q) = \{e\} \cup C = C'$. From the definition of the translation function ESPSI it is easy to see that $\text{ESPSI}(\mathcal{E}, C') = (\!|\{e\}|\!) \,||\, Q$. $\square$

Lemma 2.3.16 (transitions are reflected): For an event structure $\mathcal{E}$ and a configuration C, any transition $\text{ESPSI}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} P'$ is matched by a step $C \xrightarrow{e} C'$, with $P' = \text{ESPSI}(\mathcal{E}, C')$.

Proof: We know that for $\text{ESPSI}(\mathcal{E}, C)$ to have a transition labelled with e it must be of the form $\text{ESPSI}(\mathcal{E}, C) = P_e \,||\, Q$ where $P_e = \mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)$, with $\varphi_e = (<e, \sharp e)$. We know from Lemma 2.3.14 that the frame of $\text{ESPSI}(\mathcal{E}, C)$ is the assertion corresponding to C, which is $\mathcal{F}(P_e \,||\, Q) = \mathbf{1} \otimes \Psi_Q = \Psi_Q$. For the transition e to be enabled we also know from Definition 2.3.7 that $<e \subseteq \Psi_Q$ and $\sharp e \cap \Psi_Q = \emptyset$. From how φ_e is created in the Definition 2.3.9 we know that e must be enabled in $(\mathcal{E}, C)$. Therefore, we have the transition $(\mathcal{E}, C) \xrightarrow{e} (\mathcal{E}, C')$, where $C' = \{e\} \cup C$.

After a transition $P_e \,||\, Q \overset{e}{\rightsquigarrow} (\!|\{e\}|\!) \,||\, Q$ we have that the new frame of the process is $\{e\} \cup \Psi_Q$. From Definition 2.3.9 we see that $\text{ESPSI}(\mathcal{E}, C')$ would create an eventPsi-process where all but the sub-process for e will be the same as for $\text{ESPSI}(\mathcal{E}, C)$, and the sub-process P_e will be $(\!|\{e\}|\!)$ instead of $\mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)$. This is the same process that we got after the transition in eventPsi. $\square$

Theorem 2.3.17 (preserving interleaving diamonds): For an event structure $\mathcal{E} = (E, \leq, \sharp)$ with two concurrent events $e || e'$, then in the translation $\text{ESPSI}(\mathcal{E}, \emptyset)$ we find the behaviour forming the interleaving diamond, i.e., there exists a C s.t.

$$\text{ESPSI}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} P_1 \overset{e'}{\rightsquigarrow} P_2 \text{ and}$$

$$\text{ESPSI}(\mathcal{E}, C) \overset{e'}{\rightsquigarrow} P_3 \overset{e}{\rightsquigarrow} P_2.$$

Proof: In a prime event structure if two events e, e' are concurrent then there exists a configuration C reachable from the root which contains the conditions of both events, i.e., $<e \subseteq C$ and $<e' \subseteq C$, and does not contain any of the two events, i.e., $e, e' \notin C$. This can be seen from Remark 2.3.6(3) which ensures the existence of some configurations C_1, C_2, and $C_1 \cup C_2$, which contains both e, e'. Removing from this last configuration both e, e' we still obtain a configuration. Take this configuration as the one C sought in the theorem. Therefore we have the following steps in the event structure: $C \xrightarrow{e} C \cup \{e\}$, $C \xrightarrow{e'} C \cup \{e'\}$, $C \cup \{e\} \xrightarrow{e'} C \cup \{e, e'\}$, and $C \cup \{e'\} \xrightarrow{e} C \cup \{e, e'\}$.

Since C is reachable from the root then by Lemma 2.3.15 all the steps are preserved in the behaviour of the eventPsi-process $\text{ESPSI}(\mathcal{E}, \emptyset)$, meaning that $\text{ESPSI}(\mathcal{E}, C)$ is reachable from (i.e., part of the behaviour of) $\text{ESPSI}(\mathcal{E}, \emptyset)$.

Since $e, e' \notin C$ we have that $\textsc{espsi}(\mathcal{E}, C)$ is in the form $P_0 = P_e \,||\, P_{e'} \,||\, Q$ with P_e and $P_{e'}$ processes of kind **case**. From Lemma 2.3.14 we know that the frame of $\textsc{espsi}(\mathcal{E}, C)$ is the assertion corresponding to C, which is $\mathcal{F}(P_e \,||\, P_{e'} \,||\, Q) = \emptyset \cup \emptyset \cup \Psi_Q = \Psi_Q$.

From Lemma 2.3.15 we see the transitions between the eventPsi-processes: $\textsc{espsi}(\mathcal{E}, C) \overset{e}{\leadsto} P_1 \overset{e'}{\leadsto} P_2$ with $P_2 = (\!|\{e\}|\!) \,||\, (\!|\{e'\}|\!) \,||\, Q$ as well as $\textsc{espsi}(\mathcal{E}, C) \overset{e'}{\leadsto} P_3 \overset{e}{\leadsto} P_4$ with $P_4 = (\!|\{e\}|\!) \,||\, (\!|\{e'\}|\!) \,||\, Q$. We thus have the expected interleaving diamond.

As an aside, remark that $\mathcal{F}(P_1) = \mathcal{F}(P_0) \otimes \{e\}$ and $\mathcal{F}(P_3) = \mathcal{F}(P_0) \otimes \{e'\}$ thus $\mathcal{F}(P_1) \otimes \mathcal{F}(P_3) = \mathcal{F}(P_0) \otimes \{e\} \otimes \{e'\} = \mathcal{F}(P_4)$, which says that $e \in \mathcal{F}(P_1) \wedge e' \notin \mathcal{F}(P_1) \wedge e' \in \mathcal{F}(P_3) \wedge e \notin \mathcal{F}(P_3) \wedge \mathcal{F}(P_1) \otimes \mathcal{F}(P_3) = \mathcal{F}(P_4)$. Using Lemma 2.3.14 these can be correlated with configurations and thus we can see the definition of concurrency from configurations as in Remark 2.3.6(3). □

Intuitively the next result says that any two events that in the behaviour of the eventPsi-process make up the interleaving diamond are concurrent in the corresponding event structure.

Theorem 2.3.18 (reflecting interleaving diamonds): For any event structure $\mathcal{E}$, in the corresponding eventPsi-process $\textsc{espsi}(\mathcal{E}, \emptyset)$, for any interleaving diamond

$$\textsc{espsi}(\mathcal{E}, C) \overset{e}{\leadsto} P_1 \overset{e'}{\leadsto} P_2$$

and

$$\textsc{espsi}(\mathcal{E}, C) \overset{e'}{\leadsto} P_3 \overset{e}{\leadsto} P_2$$

for some configuration $C \in \mathbb{C}_{\mathcal{E}}$, we have that the events $e||e'$ are concurrent in $\mathcal{E}$.

Proof: Since $\textsc{espsi}(\mathcal{E}, C)$ has two outgoing transitions labelled with the events e and e' it means that $\textsc{espsi}(\mathcal{E}, C)$ is in the form $P_0 = P_e \,||\, P_{e'} \,||\, Q$ with P_e and $P_{e'}$ processes of kind **case**. From Lemma 2.3.14 we know that the frame of $\textsc{espsi}(\mathcal{E}, C)$ is the assertion corresponding to C, which is $\mathcal{F}(P_e \,||\, P_{e'} \,||\, Q) = \emptyset \cup \emptyset \cup \mathcal{F}(Q) = \Psi_Q$.

We thus have that $e, e' \notin \Psi_Q$ and $P_0 \overset{e}{\leadsto} P_1$ and $P_0 \overset{e'}{\leadsto} P_3$. This means that for these two transitions to be possible it must be that the precondition for e and e' respectively must be met. Since $e, e' \notin \Psi_Q$ it must be that $e' \notin \pi_L(\varphi_e)$ and $e \notin \pi_L(\varphi_{e'})$. Since $\pi_L(\varphi_e)$ is the same as the set $<e$ and $\pi_L(\varphi'_e)$ the set $<e'$ we have the two parts of the Definition 2.3.3 that concern $\leq$ for the causal independence (concurrency) of the events e, e', i.e., $\neg(e' \leq e \vee e \leq e')$. After the two transitions are taken we have that $P_1 = (\!|\{e\}|\!) \,||\, P_{e'} \,||\, Q$ and $P_3 = P_e \,||\, (\!|\{e'\}|\!) \,||\, Q$. We thus have that $e \in \mathcal{F}(P_1)$ and $e' \in \mathcal{F}(P_3)$. For the transition $P_1 \overset{e'}{\leadsto} P_2$ to happen we must have that $e \notin \pi_R(\varphi_{e'})$ and for $P_3 \overset{e}{\leadsto} P_4$ we must have $e' \notin \pi_R(\varphi_e)$. This is the same as $e' \notin \sharp e$ and $e \notin \sharp e'$ which makes the last part of Definition 2.3.3 concerning the conflict relation, i.e., $\neg(e' \sharp e)$. This completes the proof, showing $e||e'$. □

Theorem 2.3.19 (syntactic restrictions): For any syntactically correct eventPsi-process P_{ES} there exists an event structure $\mathcal{E}$ and a configuration $C \in \mathbb{C}_{\mathcal{E}}$ s.t.

$$\textsc{espsi}(\mathcal{E}, C) = P_{ES}.$$

Proof: From an eventPsi-process P_{ES} defined according to the syntactic restrictions of Definition 2.3.10, we show how to construct an event structure $\mathcal{E} = (E, \leq, \sharp)$ and a configuration C. We have that P_{ES} is built up of assertion processes and **case** guarded outputs, i.e.,

$$P_{ES} = (\prod_{e \in E_c} (\!|\{e\}|\!)) \quad || \quad (\prod_{f \in E_r} \textbf{case } \varphi_f : \overline{f}\langle f \rangle.(\!|\{f\}|\!)).$$

Because of the third restriction on P_{ES} (i.e., from Definition 2.3.10(3)) we know that E_c and E_r are sets, since P_{ES} cannot have in parallel two assertion processes with the same assertion, nor two **case** processes with the same channel. Moreover, these two sets are disjoint.

We take C to be the frame of $\mathcal{F}(P_{ES}) = E_c$. We take the set of events to be $E = E_c \cup E_r$. We construct the causality and conflict relations from the processes in the second part of P_{ES} as follows: $< := \cup_{e \in E_r}\{(e', e) | e' \in \pi_L(\varphi_e)\}$ and $\sharp := \cup_{e \in E_r}\{(e', e) | e' \in \pi_R(\varphi_e)\}$. We prove that the causality relation is a partial order. For irreflexivity just use the first part of the second restriction on P_{ES}. For antisymmetry assume that $e \leq e' \wedge e' \leq e \wedge e \neq e'$ which is the same as having $e \in \pi_L(\varphi_{e'}) \wedge e' \in \pi_L(\varphi_e)$. This contradicts the first part of the second restriction on P_{ES}. Transitivity is easy to obtain from the second part of the second restriction which says that when $e \leq e'$ then all the conditions of e are a subset of the conditions of e'. We prove that the conflict relation is irreflexive and symmetric. The irreflexivity follows from the first part of the first restriction on P_{ES}, whereas the symmetry is given by the second part.

It is easy to see that for the constructed event structure and the configuration chosen above, we have $\text{ESPSI}(\mathcal{E}, C) = P_{ES}^C$. The encoding function ESPSI takes all events from C to the left part of the P_{ES}, whereas the remaining events, i.e., from E_r are taken from **case** processes where for each event $f \in E_r$ the corresponding condition φ_f contains the causing events respectively the conflicting events. But these correspond to how we built the two relations above. $\square$

Notation: For a syntactically correct process P (i.e., restricted according to Definition 2.3.10) we will use the notation E_P for the events associated to the process P as defined in the above proof, i.e, $E_P = E_c \cup E_r$.

2.3.3 Action refinement

Below we show how to refine eventPsi processes corresponding to *action refinement* for labelled event structures [vGG01] as recalled below. The intuition of action refinement is to be able to give actions (which are thought of as possible abstractions) more structure, by replacing every event labelled by a particular action with a finite, conflict free event structure (with events possibly labelled by other actions). For example one action can be refined into a sequence of actions, or in general any deterministic finite concurrent process.

Since action refinement is defined using the action labels, we assume our eventPsi instances have labelled events of the form $(e, l(e))$ for some labelling function $l : E \to Act$, and as usual write e^a for an event (e, a).

A *refinement function* $ref : Act \to \mathbb{E}_{\not\sharp}$ is then a function from the set of actions of event structures (denoted by Act) to conflict-free event structures (i.e., the conflict relation is empty) denoted by $\mathbb{E}_{\not\sharp}$. The function ref is considered as a given function to be used in the *refinement operation* denoted by $\mathbf{ref}$.

Definition 2.3.20 (refinement for prime event structures):

For an event structure $\mathcal{E}$ with events labelled by $l : E \to Act$ a function $ref : Act \to \mathbb{E}_{\not\sharp}$ is called a *refinement function* (for prime event structures) iff $\forall a \in Act : ref(a)$ is a non-empty, finite and conflict-free labelled prime event structure.

For $\mathcal{E} \in \mathbb{E}$ and ref a refinement function, let $\mathbf{ref}(\mathcal{E}) \in \mathbb{E}$ be the prime event structure defined by:

- $E_{\mathbf{ref}(\mathcal{E})} := \{(e, e') | e \in E_{\mathcal{E}}, e' \in E_{ref(l_{\mathcal{E}}(e))}\}$, where $E_{ref(l_{\mathcal{E}}(e))}$ denotes the set of events of the event structure $ref(l_{\mathcal{E}}(e))$,
- $(d, d') \leq_{\mathbf{ref}(\mathcal{E})} (e, e')$ iff $d \leq_{\mathcal{E}} e$ or $(d = e \wedge d' \leq_{ref(l_{\mathcal{E}}(d))} e')$,
- $(d, d') \sharp_{\mathbf{ref}(\mathcal{E})} (e, e')$ iff $d \sharp_{\mathcal{E}} e$,
- $l_{\mathbf{ref}(\mathcal{E})}(e, e') := l_{ref(l_{\mathcal{E}}(e))}(e')$.

Example 2.3.21: Figure 2.4(a) provides an illustrative example (taken from [vGG01]) of action refinement applied to a process which receives and sends data. We may think of giving more details to the sending event by refining it with a sequential process which first prepares the data and then carries out the actual sending. This refined process is shown in Figure 2.4(b). In turn, Figure 2.4(c) shows a further refinement, where the preparation of data is refined by a process which in parallel formats the data and asks permission to send.

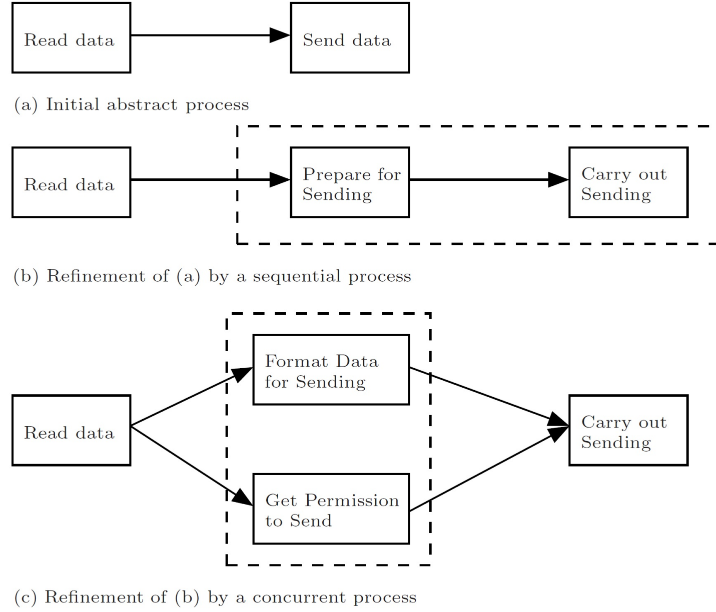


Fig. 2.4: Example for action refinement from [vGG01].

Remark 2.3.22: Action refinement as presented here was introduced by Wirth [Wir71] under the name of *stepwise refinement* and is quite different than the more recent notion of refinement in process algebras where the refined process is seen as an *implementation* of the abstract one. In these settings usually refinement is seen as an inclusion-like relation of the behaviours, such as trace inclusion or simulation.

We can define a refinement function $ref^{\psi} : Act \to \mathbf{P}_{\mathsf{eventPsi}}$ for eventPsi-processes from a given refinement function ref for event structures as follows: $ref^{\psi}(a) = \textsc{espsi}(ref(a))$. For syntactically correct eventPsi-processes these two functions are one-to-one inter-definable in the sense that for eventPsi-processes that represent finite conflict-free event structures (i.e., that have the right elements of the conditions always empty) we can define ref from ref^{ψ} by an analogous definition as before, going through Theorem 2.3.19. In the rest of this section, we prefer to work with the simpler notation provided by ref.

We define an action refinement operation for eventPsi-process terms.

Definition 2.3.23: Given a refinement function ref for event structures over events E, we define an operation $\mathbf{ref}^{\Psi}$ that refines an eventPsi-process over events E to a new one over the terms

$$\mathbf{T}^{\Psi} = \{(e, e') \mid e \in \mathbf{T}, e' \in E_{ref(l(e))}\}.$$

A syntactically correct eventPsi-process P, built according to Definition 2.3.10, with frame $\mathcal{F}(P) = \Psi_P$, is refined into a process

$$\mathbf{ref}^{\Psi}(P) = \prod_{(e,e') \in \mathbf{T}^{\Psi}} P_{(e,e')},$$

and

$$P_{(e,e')} = \begin{cases} (\!|\{(e,e')\}|\!) & \text{if } e \in \Psi_P \\ \mathbf{case}\ \varphi_{(e,e')} : \overline{(e,e')}\langle(e,e')\rangle.(\!|\{(e,e')\}|\!) & \text{otherwise} \end{cases}$$

with the conditions being

$$\varphi_{(e,e')} = (<(e,e'), \sharp(e,e')),$$

where

$$<\!(e,e') = \{(d,d') \mid d \in \pi_L(\varphi_e) \vee (d = e \wedge d' <_{ref(l(d))} e')\}$$

and

$$\sharp(e,e') = \{(d,d') | d \in \pi_R(\varphi_e)\}.$$

The set of events (which constitute the terms) is the set of pairs of an event from the original process and one of the events from the refinement processes. Note that the above definition is still part of the eventPsi instance because we can map $\mathbf{T}^\Psi$ onto $\mathbf{T}$. Take any total order $<$ on E and define from it a total order $(e,e') < (d,d')$ iff $e < d \vee (e = d \wedge e' < d')$ on the pairs; map any pair to an event from E while preserving the order, thus making $\mathbf{T}^\Psi$ the same as the $\mathbf{T}$ of eventPsi.

We make new conditions for each event (e_1, e_2), where $<\!(e_1,e_2)$ contains all pairs of events (e_1', e_2') s.t. either $e_1' < e_1$, or $e_1' = e_1 \wedge e_2' < e_2$. We define conflicts by $\sharp(e_1,e_2) = \{(e_1',e_2') \mid e_1 \# e_1'\}$ (recalling that the refinement process is conflict-free). The refinement generates for each new pair one process which is either an assertion or a **case** process, depending on whether the first part of the event pair was in the frame of the old P respectively not.

Theorem 2.3.24 (refinement in eventPsi corresponds to that in ES):

For any prime event structure $\mathcal{E}$ we have that:

$$\textsc{espsi}(\mathbf{ref}(\mathcal{E}), \emptyset) = \mathbf{ref}^\Psi(\textsc{espsi}(\mathcal{E}, \emptyset)).$$

Proof: As $\mathbf{T} = E$ and $\mathbf{T}^\Psi$ is built from $\mathbf{T}$ in Definition 2.3.23 with rules analogous to those E_{ref} is built from E in Definition 2.3.20, we have that $\mathbf{T}^\Psi = E_{ref}$. Since the processes we work with are parallel compositions of assertion and **case** processes, it means we have to show that any assertion processes on the left is also found on the right of the equality (and vice versa), and the same for the **case** processes. Since we work with the empty initial configuration, then there are no assertion processes on either sides.

The **case** processes on the left side of the equality are those generated by ESPSI from the pairs of events returned by the **ref** from the event structure $\mathcal{E}$, i.e., $P_{(e,e')} = \mathbf{case}\ \varphi_{(e,e')} : \overline{(e,e')}\langle(e,e')\rangle.(\!|\{(e,e')\}|\!)$ with the condition $\varphi_{(e,e')} = (<\!(e,e'), \sharp(e,e'))$ which is build according to Definition 2.3.20 from $\sharp_{\mathcal{E}} e$, $<_{\mathcal{E}} e$, and $<_{ref(l(d))} e'$. On the right side we have **case** processes for the original process before the refinement, with their respective conditions. But the $\mathbf{ref}^\Psi$ replaces each one of these P_e with several **case** processes, one for each new pair (e,e') that involves e. Therefore, according to Definition 2.3.23 we will have $P_{(e,e')} = \mathbf{case}\ \varphi_{(e,e')} : \overline{(e,e')}\langle(e,e')\rangle.(\!|\{(e,e')\}|\!)$ with its condition being built from $\pi_R(\varphi_e)$, $\pi_L(\varphi_e)$, and $<_{ref(l(d))} e'$. These are respectively the same as the ones used on the left side. Checking that any case process from the right side is found in the left side is done similarly. ☐

In this section we gave a representation of an encoding of the prime event structures into an instance of Psi-calculi which preserves the causality relation and thereby also the notion of action refinement. To do this, we made special use of the logic of Psi-calculi, i.e., of the assertions and conditions and the entailment between these, as well as the assertion processes. It is noteworthy that we have not used neither names nor the communication mechanism of Psi-calculi, which is known to increase expressiveness.[6]

2.4 DCR-graphs as Psi-calculi

2.4.1 Background on DCR-graphs

Dynamic Condition Response graphs (DCR-graphs) [HM10, HMS12] is a model of concurrency which generalises event structures in two dimensions: Firstly, it allows finite models of (regular) infinite behaviour, while retaining the possibility of infinite models. The finite models are regular in the automata-theoretic sense, i.e. they (if concurrency is ignored) capture exactly the languages that are the union of a regular and an omega-regular language [DHS15a]. Finite DCR-graphs have found applications in practice for the description, implementation and automated verification of flexible workflow systems [Sla15, SMHM13]. Infinite DCR-graphs allow for representation of non-regular

[6] In pi-calculus communication of channel names allows to reach Turing completeness [Mil92], in contrast to CCS with only synchronization and replication (called $\text{CCS}_!$) where the expressiveness is weaker [AGNV07].

behaviour and denotational semantics. Secondly, the DCR-graphs model provides an event-based notion of acceptance criteria for both finite and infinite computations in terms of scheduled responses. In the present chapter we focus on the first dimension, leaving the interesting question of representing event-based acceptance criteria for infinite computations to future work. We follow the notations for DCR-graphs from [HM10, HMS12].

Definition 2.4.1 (DCR-Graphs): We define a *Dynamic Condition Response Graph* to be a tuple $\mathcal{D} = (E, M, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%, L, l)$ where

1. E is a set of events,
2. $M \in 2^E \times 2^E \times 2^E$ is the initial marking,
3. $\rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\% \subseteq E \times E$ are respectively called the *condition*, *response*, *milestone*, *include*, and *exclude* relations,
4. $l : E \rightarrow L$ is a *labelling function* mapping events to labels taken from L.

For any relation $\rightarrow \in \{\rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%\}$, we use the notation $e\!\rightarrow$ for the set $\{e' \in E \mid e \rightarrow e'\}$ and $\rightarrow\! e$ for the set $\{e' \in E \mid e' \rightarrow e\}$.

A *marking* $M = (Ex, Re, In)$ represents a state of the DCR-graph. One should understand Ex as the set of *executed* events, Re the set of scheduled *response* events[7] that must happen sometime in the future or become excluded for the run to be accepting (see Definition 2.4.4), and In the set of currently *included* events. The five relations impose constraints on the events and dictate the dynamic inclusion and exclusion of events.

Intuitively, the condition relation $e \rightarrow\!\!\bullet\, e'$ requires the event e to have happened (at least once) or currently be excluded in order for e' to happen. The response relation $e \bullet\!\!\rightarrow e'$ means that if the event e happens, then the event e' becomes scheduled as a response. The milestone relation $e \rightarrow\!\!\diamond\, e'$ imposes the constraint that e' cannot happen as long as e is a scheduled response and included. Finally, the exclusion and inclusion relations generalize the conflict relation from event structures. An event e that excludes another event e' can be thought as being in (one-sided) conflict; but another event may include e' again, thus making the previous conflict only *transient*.

An event is thus *enabled* if it is included, all its included preconditions have been executed, and none of the included events that are milestones for it are scheduled responses. In particular, an event can happen an arbitrary number of times as long as it is enabled. We express the enabling condition formally as follows.

Definition 2.4.2 (enabling events): For a DCR-graph $\mathcal{D} = (E, M, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%)$ with an initial marking $M = (Ex, Re, In)$, we say that an *event* $e \in E$ *is enabled in* M, written $M \vdash e$, iff

$$e \in In \;\wedge\; (In \cap \rightarrow\!\!\bullet e) \subseteq Ex \;\wedge\; (In \cap \rightarrow\!\!\diamond e) \subseteq (E \setminus Re).$$

Having defined when events are enabled, we can define an event labelled transition semantics for DCR-graphs. Since the execution of an event only changes the marking, we define the transition relation between the markings of a given DCR-graph and regard the marking M given in the DCR-graph as the initial marking.

Definition 2.4.3 (transitions): The behaviour of a DCR-graph is given through *transitions between markings* done by executing enabled events. The result of the execution in a DCR-graph $\mathcal{D} = (E, M_0, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%)$ from marking $M = (Ex, Re, In)$ of an enabled event $M \vdash e$ results in the new marking

$$M' \stackrel{def}{=} (Ex \cup \{e\}, (Re \setminus \{e\}) \cup e\bullet\!\!\rightarrow, (In \setminus e\!\rightarrow\!\!\%) \cup e\!\rightarrow\!\!+).$$

and is written as the e-labelled transition $M \xrightarrow{e} M'$. The (interleaving) semantics of the DCR-graph is then defined as the event-labelled transition system with markings as states and M_0 the initial state.

We can now define (possibly infinite) runs of DCR-graphs and the acceptance criteria formally. As stated above, every event scheduled as response must either happen or be excluded in the future, in order for the run to be accepting. An event is no longer scheduled as a response after it has happened, unless it is related to itself by a response relation.

[7] similar to the notion of restless events in [Win80, ch.6.4.].

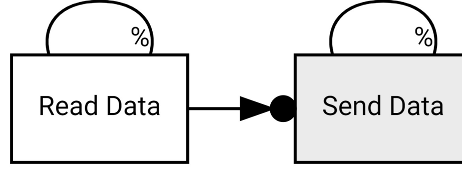


Fig. 2.5: Simple DCR-graph of a read-send process

Definition 2.4.4 (accepting runs [HM10]): A *run* of a DCR-graph with initial marking M_0 is a (possibly infinite) sequence of transitions $M_i \xrightarrow{e_i} M_{i+1}$, with $0 \leq i < k$, $k \in \mathbb{N} \cup \{\omega\}$, and $M_i = (Ex_i, Re_i, In_i)$. A run of a DCR-graph is *accepting* (or completed) if it holds that

$$\forall i \geq 0, e \in Re_i . \exists j \geq i : (e = e_j \vee e \notin In_j)$$

In words, a run of a DCR-graph is accepting if no event scheduled as an response is included and pending forever without happening, i.e. it must either eventually happen on the run or become excluded.

Since in Psi-calculi we do not have readily available a notion of accepting run, we will ignore this aspect for the rest of this chapter. However, since our encoding captures the response and milestone relations of DCR-graphs, it is prepared for adding a notion of accepting runs via scheduled responses to Psi-calculi.

It is worth noting that the labelling function on events adds the possibility of non-determinism by taking the language of a DCR-graph to be the sequences of labels of events (abstracting from the events) of accepting runs. This extra level of labelling increases the expressive power of finite DCR-graphs, which have been shown to capture exactly the languages that are the union of a regular and an omega-regular language [DHS15a]. In contrast, by following an encoding along the lines of [RHT08] unlabelled, finite DCR-graphs can be represented by Linear-time Temporal Logic (LTL), which is known to be strictly less expressible than omega-regular languages.

As given by the mapping in Definition 2.4.5 below, prime event structure can be seen as a special case of a DCR-graph (see [HM10, Prop.1&3] for details) where the exclusion relation (capturing the conflict relation) is reflexive and symmetric, the condition relation (capturing the causality relation) is irreflexive and transitive, and the include, response, and milestone relations are empty. The initial marking has no executed events, no scheduled responses and all events are included. Hereto comes of course the two conditions on the causality and conflict relation of event structures, i.e. finite causes and hereditary conflict. The reflexivity of the exclusion relation and emptiness of the inclusion relation imply that events can be executed at most once.

Definition 2.4.5 (prime event structures as DCR-graphs [HM10]): Define a mapping **dcr** which takes an event structure $\mathcal{E} = (E, \leq, \sharp, l)$ and returns its presentation as a DCR-graph $(E, M, <\ , \emptyset, \emptyset, \emptyset, \sharp \cup \{(e,e) \mid e \in E\})$ with the marking $M = (\emptyset, \emptyset, E)$.

Example 2.4.6: Consider the small DCR-graph $\mathcal{D}$ shown in Figure 2.5, corresponding to the event structure of Figure 1.6(a) which was representing a process of first reading data (R) and then sending data (S). The DCR-graph is formalised as

$$\mathcal{D}_1 = (\{\text{R}, \text{S}\}, (\emptyset, \emptyset, \{\text{R}, \text{S}\}), \{(\text{R}, \text{S})\}, \emptyset, \emptyset, \emptyset, \{(\text{R}, \text{R}), (\text{S}, \text{S})\}).$$

Here we can see that each event removes itself from the included set when it happens, and that for S to happen its prerequisite R must happen. This corresponds to the causality relation in the event structure in Figure 1.6(a).

In DCR graphs is possible to demand that if we read some data we will eventually send this data. This is modelled in the DCR-graph of Figure 2.6 formalised as

$$\mathcal{D}_2 = (\{\text{R}, \text{S}\}, (\emptyset, \emptyset, \{\text{R}, \text{S}\}), \{(\text{R}, \text{S})\}, \{(\text{R}, \text{S})\}, \emptyset, \emptyset, \{(\text{R}, \text{R}), (\text{S}, \text{S})\}),$$

where we added a response relation from R to S. This means that a run is only accepting if any R is eventually followed by an S, e.g., the empty run and the run R.S are accepting, while the run consisting of the single event R is not.

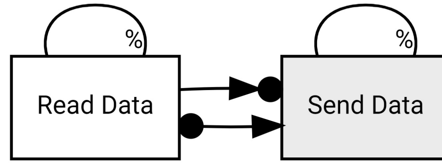


Fig. 2.6: DCR-graph with added responses

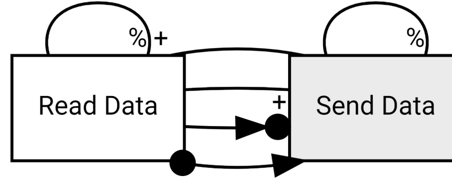


Fig. 2.7: Message forwarder DCR-graph with possible infinite execution.

The examples above only allow each event to happen once. However, as exemplified below, in DCR-graphs the set of events needed for an event to be enabled can change during the run, as events are included or excluded. Moreover, the conflict in DCR-graphs is not permanent as is the case with event structures. Conflict in DCR-graphs is *transient* since an event can be included and later excluded during a run. So, already at the conflict and causality relations, the DCR-graphs depart from event structures in a non-trivial manner.

Example 2.4.7: A message forwarding machine, where the events can happen several times, but alternating, can be represented by the DCR-graph in Figure 2.7 formalised as

$$\mathcal{D}_3 = (\{\textsc{r}, \textsc{s}\}, (\emptyset, \emptyset, \{\textsc{r}, \textsc{s}\}), \{(\textsc{r}, \textsc{s})\}, \{(\textsc{r}, \textsc{s})\}, \emptyset, \{(\textsc{r}, \textsc{s}), (\textsc{s}, \textsc{r})\}, \{(\textsc{r}, \textsc{r}), (\textsc{s}, \textsc{s})\}).$$

Each time one of the two events happens it excludes itself but includes the other event, modelling the alternation between reading and sending. We still have that if R happens, eventually S must happen for the run to be accepting, but we make no requirements on how many times the events can happen, so the unique infinite execution and every finite execution of even length will be accepting.

2.4.2 Encoding DCR-graphs

Below we provide an encoding of finite DCR-graphs as a Psi-calculi instance. As was done with configurations of event structures, markings are kept in the frame of the process. Also, similarly to the event structure representation, for each event of the DCR-graph we use a **case** process, and conditions and entailment relation to capture the information needed to decide when events of a DCR graph are enabled in a marking.

However, in contrast to the encoding of event structures, it appears that we need the communication constructs on processes to keep track of the *current marking* of a DCR-graph. The expressiveness of DCR-graphs seems not to allow for a simple way of updating the marking, as was the case for event structures where union with the newly executed event was enough. But once we use the communication we get a nice natural encoding for DCR-graphs in a Psi-calculus instance. The idea is to internally communicate a term representing the current marking, and incorporate a generation (or age) of an assertion and then keep the assertion with the latest generation when composing assertions. Generations are inspired by [BHJ$^+$11], where they are used in a similar way to represent the changing topology of a mobile communication network.

Since a step is now an internal communication, we can no longer observe the event that happened by just looking at the communication channel. Instead, we record all executed events in a multi-set part of the assertion. The

underlying set of this multi-set corresponds to the set of executed events in the marking of the DCR-graph. By recording also the multiplicity of each event one can identify, by multi-set subtraction, which event happened in a transition, as expressed formally below.

Notation 2.4.8: For a multi-set S we will denote its underlying set by $\lfloor S \rfloor$ and write $|S|$ for the number of elements of S, summing multiplicities.

Definition 2.4.9 (dcrPsi instance): Given a set of *constants* E (denoting events), we define the dcrPsi instance over a set of names $\mathcal{N}$ as:

$$\mathbf{T} \overset{def}{=} \mathcal{N} \cup \mathbf{A} \cup \mathbb{N}^E_{op} \cup 2^E_{op} \cup \mathbb{N}_{op}$$

$$\mathbf{A} \overset{def}{=} (\mathbb{N}^E_{op} \times 2^E_{op} \times 2^E_{op} \times \mathbb{N}_{op}) \cup \mathbf{1} \qquad \mathbf{1} \overset{def}{=} \bot$$

where $\mathbb{N}^E_{op}$ are the multi-sets over E with the operators of Example 2.2.19 and possibly containing names from $\mathcal{N}$, $\mathbb{N}_{op}$ is the data structure capturing natural numbers from Example 2.2.18, and $\bot$ is a special constant term assertion.

(We will refer to assertions containing only ground terms as ground assertions, and assertions containing names as open assertions.)

$$\mathbf{C} \overset{def}{=} (2^E_{op} \times 2^E_{op} \times E) \cup \mathbb{N}_{op} \cup \{M \leftrightarrow N \mid M, N \in \mathbf{T}\}.$$

For ground assertions, i.e., when Ex, Re, In, g denote ground terms, define

$$(Ex, Re, In, g) \otimes (Ex', Re', In', g') \overset{def}{=} \begin{cases} (Ex, Re, In, g) & \text{if } g > g', \\ (Ex', Re', In', g') & \text{if } g < g', \\ (\emptyset, \emptyset, \emptyset, g) & \text{if } g = g', \end{cases}$$

where the comparison $g < g'$ is done using sub-term relation, e.g., $s(g) > g$ (we usually denote generations by $g, k \in \mathbb{N}$). For the other cases define

$$\Psi_a \otimes \Psi_b \overset{def}{=} \begin{cases} \Psi_a & \text{when } \Psi_b \text{ is open or } \mathbf{1}, \text{ and } \Psi_a \text{ is ground}, \\ \Psi_b & \text{when } \Psi_a \text{ is open or } \mathbf{1}, \text{ and } \Psi_b \text{ is ground}, \\ \mathbf{1} & \text{otherwise}. \end{cases}$$

Entailment $\vdash$ is defined as:

$$(Ex, Re, In, g) \vdash (Co, Mi, e) \text{ iff } e \in In \wedge (In \cap Co) \subseteq Ex \wedge ((In \cap Mi) \cap Re) = \emptyset$$

$$(Ex, Re, In, k) \vdash g \in \mathbb{N} \quad \text{iff} \quad k = g$$

$$(Ex, Re, In, g) \vdash a \leftrightarrow b \ \text{ iff } \ a, b \in \mathcal{N} \text{ and } a = b.$$

For any other assertions (e.g. the open ones) or conditions the entailment is undefined.

Notation 2.4.10: We denote ground assertions, and the respective four ground terms, by (Ex, Re, In, g). In the few cases where we need to talk about open terms we will use capital letters X_E, X_R, X_I, X_G possibly indexed to indicate the respective component in the tuple, to stand for a name. We do not always mention explicitly if the assertions or terms are ground when this is clear from the context or the notation.

Terms can be either a name or assertions (and their components) which will be the data communicated. Assertions are four-tuples of one multi-set and two sets containing events, whereas the fourth element is a number which we intend to hold the *generation* of the assertion.[8] The multi-sets capture which events have been executed, and also count how many times each event has been executed. This will allow, in Definition 2.4.23, to infer from the

[8] For all these terms we allow operators to be present, but for simplicity we work with their mathematical presentation, and assume that when evaluated the equations of the respective operators would produce the ground final form, like numbers, sets, multi-sets.

change in the frame, which event happened in an execution step. The second set holds those events that are pending responses, and the third set those events that are included. The multi-set and set operations in the terms allow us to construct terms for updating the sets when an event is executed. The underlying set of the multi-set represents the first set of a marking from DCRs, whereas the two other sets of an assertion represent the second and third set of a marking of a DCR-graph. The generation number helps to get the properties of the assertion composition, which are somewhat symmetric, but still have the composition return only the latest marking/assertion (i.e., somewhat asymmetric).

The composition of two assertions keeps the assertion with highest generation if both are tuples of ground terms. For technical reasons, when we compose two ground assertions with the same generation we obtain an assertion where the first three elements are emptysets, and the generation number remains unchanged. Intuitively, we do not want two assertions with the same generation number to exist because this can be thought as an error in the process computation (i.e., assertion generations are supposed to constantly increase). But in our encodings and results this never happens, which can be easily checked. Moreover, technically it is a nice solution to make any two assertions with the same generation disappear.

When any of the assertions contain names (i.e., either one of the tuple elements is a name or a term contains a name) then the composition returns the ground assertion when it exists or the identity assertion, when both assertions contain names. In other words, we are interested only in the ground assertions, those containing the actual data, without undefined parts. This makes the composition operation associative, commutative, compositional wrt. assertion equivalence,[9] and with identity being the special constant term $\bot$.

The conditions are tuples of two sets of events and a single event as the third tuple component. The first set is intended to capture the set of events that are conditions for the single event. The second set is intended to capture the set of events that are milestones for the single event.

As in [BHJ+11], we added the set of natural numbers as conditions for technical reasons, so that assertion equivalence will be compositional. Without this we would have $(Ex, Re, In, 0) \simeq (Ex, Re, In, 2)$, but when composed with another assertion $(Ex', Re', In', 1)$, where $In \cap In' = \emptyset$, we would get $(Ex, Re, In, 0) \otimes (Ex', Re', In', 1) \not\simeq (Ex, Re, In, 2) \otimes (Ex', Re', In', 1)$ which contradicts compositionality: On the left side we would keep the assertion with highest generation 1 whereas on the right side we would keep the one with 2. As the sets of *included* events are different we have that they cannot entail the same conditions. With the natural numbers as conditions we can distinguish between the two assertions $(Ex, Re, In, 0) \not\simeq (Ex, Re, In, 2)$ since we have that $(Ex, Re, In, 0) \nvdash 2$ but $(Ex, Re, In, 2) \vdash 2$.

Remark 2.4.11 (on sorting): We rely on sorting to properly define in dcrPsi the terms, substitutions, matching, etc. We rarely mention sorting aspects, only when necessary for clarifications, and rely on the intuition in most cases. For instance, in Definition 2.4.9 we silently use sorts for several aspects: to distinguish the different kinds of terms (like a sort for natural numbers, another sort for multi-sets); several corresponding name sorts; the assertion tuple operand is multi-sorted; the substitution takes care that names on the respective place in a tuple are replaced by terms of respective sort. We thus make careful use of notation to be in accordance with the sorting aspects; e.g., the notation (a, a, a, b) is unacceptable, opposed to (a, c, c, b) which allows both second and third elements of a tuple to be the same term.

Lemma 2.4.12: For two assertions $\Psi = (Ex, Re, In, g), \Psi' = (Ex', Re', In', g')$ we have that $\Psi \simeq \Psi' \Rightarrow g = g'$

Proof: Since $\Psi \simeq \Psi'$ and $\Psi \vdash g$ then also $\Psi' \vdash g$ which means that $g' = g$. □

Lemma 2.4.13 (correctness of dcrPsi): The dcrPsi instance fulfils the requirements of being a Psi-calculi instance, cf. Definition 2.2.2 and 2.2.4.

Proof: We have to show that the channel equivalence and the composition of assertions conform with the requirements from Definition 2.2.4. We also need to show that the operators and nominal datatypes of Definition 2.2.2 are well defined.

[9] Recalling from Section 2.2, compositionality refers to: $\Psi \simeq \Psi' \Rightarrow \Psi'' \otimes \Psi \simeq \Psi'' \otimes \Psi'$.

For the channel equivalence it is easy to see that symmetry and transitivity are respected since in Definition 2.4.9(last line) channel equivalence is defined in terms of name equality.

For assertion composition we have to look at three different scenarios, one where all the assertions are open or **1**, one where we have a mix of open or **1** and ground assertions, and last where we have only ground assertions.

We first point out that all open assertions are assertion equivalent through the fact that they cannot entail any conditions (i.e., entailment in Definition 2.4.9 is undefined for open assertions). The same goes for **1** which also does not entail any conditions, and thus assertion equivalent with all open assertions. Whenever we compose two open assertions we obtain **1**, i.e., composition of open assertions results in the identity assertion that is assertion equivalent with all open assertions. (From here on we will refer to **1** as an open assertion for the sake of simplicity because we only care about assertion equivalence for the correctness proofs). Since the four requirements from Definition 2.4.9 on assertion composition are defined in terms of assertion equivalence, then they are trivially satisfied when only open assertions are involved.[10]

For the case where we have a mix of open and ground assertions, the composition returns the ground assertion (i.e., open assertions are absorbed by any ground ones). It is easy to check the four requirements from Definition 2.4.9. Also note that it is not possible for any ground assertion to be equivalent with any open assertion since the ground assertion will at least entail the condition g when is its generation.

When we have only ground assertions the composition will maintain the one with the highest generation, when composing assertions with different generations; otherwise, if we have two ground assertions with the same generation g we obtain the assertion $(\emptyset, \emptyset, \emptyset, g)$.

For commutativity it is easy to see that when the generations are the same we will always get the same assertion, independent of the order we compose them. When the assertions have different generations, keeping the one with the highest generation is independent of its place in the composition.

For compositionality we know, by Lemma 2.4.12, that if two assertions Ψ, Ψ' are equivalent then they have the same generation g. Therefore, when composed with another assertion Ψ'' with generation g'' we must treat three cases. If $g < g''$ then for both $\Psi \otimes \Psi''$ and $\Psi' \otimes \Psi''$ we obtain Ψ''. If $g = g''$ then on both sides we obtain the assertion $(\emptyset, \emptyset, \emptyset, g)$. When $g > g''$ we have that $\Psi \otimes \Psi'' = \Psi$ and $\Psi' \otimes \Psi'' = \Psi'$ which are equivalent by assumption.

For associativity, when all the generations are different we remain with the assertion that has the highest generation (i.e., by virtue of the associativity of the *max* function on natural numbers). When two or more assertions have the same generation g the result depends on whether this is the largest generation or not. When g is the largest we obtain the assertion $(\emptyset, \emptyset, \emptyset, g)$. Otherwise, the assertion with highest generation will be returned, on both sides.

To make sure that the nominal datatypes and operators of the instance are well defined consider the following observations. Assertion composition always returns a correct assertion term, whereas channel equality operation always returns a condition by virtue of the definition including all terms $M \dot{\leftrightarrow} N$.

It is not difficult to see that $\mathbf{T}, \mathbf{A}$ and $\mathbf{C}$ are nominal datatypes when the correct sorting is used. In particular, assertions are four-tuple terms that can take a name of sort multiset, two names of sort set, and one of sort natural numbers; each of which can be substituted with terms of corresponding data sorts. All these terms have finite support since they are finite. Terms can be either names, assertion tuples, or the individual terms that can be used by the substitution functions. There are then closed under name swapping, as well as under substitutions that respect the sorting discipline. □

As stated in the lemma below, the entailment mimics the definition in DCR graphs for when an event (i.e., the third component of the conditions) is enabled in a marking (i.e., the first three components of the assertions).

Lemma 2.4.14 (correlation between entailment in DCRs and dcrPsi): For any DCR graph $\mathcal{D}$ we have that

$$(Hi, Re, In, g) \vdash (\rightarrow\!\bullet e, \rightarrow\!\diamond e, e) \quad \text{iff} \quad (\lfloor Hi \rfloor, Re, In) \vdash e \quad .$$

Proof: Follows directly from Definition 2.4.2 and Definition 2.4.9 □

[10] For example the *identity* is satisfied since $\Psi \otimes \mathbf{1} \stackrel{def}{=} \mathbf{1}$ and $\mathbf{1} \simeq \Psi$ when Ψ is open.

We are now ready to provide the mapping of DCR-graphs to dcrPsi. To facilitate proving the semantical correspondence we provide a slightly more general mapping that takes a DCR-graph and a multi-set history of executed events.

Definition 2.4.15 (DCR-graphs to dcrPsi): We define a function DCRPSI($\mathcal{D}, Hi$) which takes a DCR $\mathcal{D} = (E, M \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$, with initial marking $M = (Ex, Re, In)$, and a multi-set Hi representing the history of events that have happened, with underlying set $\lfloor Hi \rfloor = Ex$, and returns a dcrPsi process

$$P_{dcr} = (\nu m)(P_k \,||\, P_E)$$

where

$$P_k = (\!|(Hi, Re, In, |Hi|)|\!) \,||\, \overline{m}\langle (Hi, Re, In, |Hi|)\rangle.\mathbf{0}$$

and

$$P_E \;=\; \prod_{e \in E} P_e$$

with

$$\begin{aligned} P_e \;=\; !(\; &\mathbf{case}\; \varphi_e : \underline{m}\langle (X_E, X_R, X_I, X_G)\rangle. \\ &(\overline{m}\langle (X_E + \{e\}, (X_R \setminus \{e\}) \cup e\bullet\!\rightarrow, (X_I \setminus e\rightarrow\!\%) \cup e\rightarrow\!+, s(X_G))\rangle.\mathbf{0} \,|| \\ &(\!|(X_E + \{e\}, (X_R \setminus \{e\}) \cup e\bullet\!\rightarrow, (X_I \setminus e\rightarrow\!\%) \cup e\rightarrow\!+, s(X_G))|\!)\,)\end{aligned}$$

where $\varphi_e = (\rightarrow\!\bullet e, \rightarrow\!\diamond e, e)$ and X_E, X_R, X_I, X_G are names.

The process P_{dcr} resulting from DCRPSI contains the process P_k that models the initial marking of the encoded DCR-graph as an assertion process, and also communicates this assertion on the channel name m. We give this assertion the generation $|Hi|$. The rest of the process, i.e., P_E, captures the events and relations of the DCR-graph as a parallel composition of processes P_e for each of the events of the encoded DCR-graph. We will write DCRPSI($\mathcal{D}$) for DCRPSI($\mathcal{D}, Hi$), when $Hi = Ex$.

Each event is encoded, following the ideas for event structures, using the **case** construct with a single guard φ_e. The guard contains the information for the event e that needs to be checked against the current marking (i.e., the assertion) to decide if the event is enabled: The set of events that are prerequisites for e (i.e., $\rightarrow\!\bullet e$) and must either be executed or excluded, the set of milestones related to e (i.e., $\rightarrow\!\diamond e$) that must either be excluded or not be scheduled as responses, and the event e itself, that must be included. The events in a DCR-graph can happen multiple times, hence the use of the replication operation as the outermost operator.

As for event structures, there may be several events enabled by a marking, hence several of the parallel **case** processes may have their guards entailed by the current assertion. Only one of these input actions will communicate with the single output action on m, and will receive in the four variables the current marking. After the communication, the input process will leave behind an assertion process containing an updated marking, and also a process ready to output on m this updated marking. In fact, after a communication, what is left behind is something looking like a P_k process, but with an updated marking and an increased generation number. The updating of the marking follows the same definition from the DCR-graphs. We also guard the channel name m so that no other input or output transitions can happen on this channel, except the internal communications.

Notation 2.4.16: In the context of Definition 2.4.15, i.e., when encoding a DCR-graph through the DCRPSI, we will use the following shorthand notation to stand for the often and similar way of updating a marking:

$$\mathsf{U}_e(X_E, X_R, X_I, X_G) \stackrel{def}{=} (X_E + \{e\}, (X_R \setminus \{e\}) \cup e\bullet\!\rightarrow, (X_I \setminus e\rightarrow\!\%) \cup e\rightarrow\!+, s(X_G)).$$

The updating notation is parametrized by an event e which is enough to extract the sets of events that are used in the denoted term. The four names can be substituted as in any term, thus a substitution

$$\mathsf{U}_e(X_E, X_R, X_I, X_G)[X_E := Ex, X_R := Re, X_I := In, X_G := g]$$

would produce the term

$$(Ex + \{e\}, (Re \setminus \{e\}) \cup e\bullet\!\rightarrow, (In \setminus e\rightarrow\!\%) \cup e\rightarrow\!+, s(g)),$$

and we usually just write $\mathsf{U}_e(Ex, Re, In, g)$.

Example 2.4.17: Taking the DCR-graph $\mathcal{D}_1$ from Figure 2.5, we can create a dcrPsi-process $P = \textsc{dcrpsi}(\mathcal{D}_1, \emptyset)$, with initial generation $|\emptyset| = 0$, as follows

$$\begin{gathered} P = (\nu m)((\!|(\emptyset,\emptyset,\{\mathrm{R},\mathrm{S}\},0)|\!) \,||\, \overline{m}\langle(\emptyset,\emptyset,\{\mathrm{R},\mathrm{S}\},0)\rangle.\mathbf{0} \,|| \\ !(\mathbf{case}\ (\emptyset,\emptyset,\mathrm{R}) : \underline{m}\langle(X_E,X_R,X_I,X_G)\rangle. \\ ((\!|\mathsf{U}_{\mathrm{R}}(X_E,X_R,X_I,X_G)|\!) \,||\, \overline{m}\langle \mathsf{U}_{\mathrm{R}}(X_E,X_R,X_I,X_G)\rangle.\mathbf{0})) \,|| \\ !(\mathbf{case}\ (\{\mathrm{R}\},\emptyset,\mathrm{S}) : \underline{m}\langle(X_E,X_R,X_I,X_G)\rangle. \\ ((\!|\mathsf{U}_{\mathrm{S}}(X_E,X_R,X_I,X_G)|\!) \,||\, \overline{m}\langle \mathsf{U}_{\mathrm{S}}(X_E,X_R,X_I,X_G)\rangle.\mathbf{0}))). \end{gathered}$$

From the entailment we can see that this process may only have a synchronisation between the output $\overline{m}\langle(\emptyset,\emptyset,\{\mathrm{R},\mathrm{S}\},0)\rangle.\mathbf{0}$ and the input guarded by the **case** $(\emptyset,\emptyset,\mathrm{R})$, making a transition $P \xrightarrow{\tau} P'$ with

$$\begin{gathered} P' = (\nu m)((\!|(\emptyset,\emptyset,\{\mathrm{R},\mathrm{S}\},0)|\!) \,||\, \mathbf{0} \,|| \\ (\!|(\{\mathrm{R}\},\{\mathrm{S}\},\{\mathrm{S}\},1)|\!) \,||\, \overline{m}\langle(\{\mathrm{R}\},\{\mathrm{S}\},\{\mathrm{S}\},1)\rangle.\mathbf{0} \,|| \\ !(\mathbf{case}\ (\emptyset,\emptyset,\mathrm{R}) : \underline{m}\langle(X_E,X_R,X_I,X_G)\rangle. \\ ((\!|\mathsf{U}_{\mathrm{R}}(X_E,X_R,X_I,X_G)|\!) \,||\, \overline{m}\langle \mathsf{U}_{\mathrm{R}}(X_E,X_R,X_I,X_G)\rangle.\mathbf{0})) \,|| \\ !(\mathbf{case}\ (\{\mathrm{R}\},\emptyset,\mathrm{S}) : \underline{m}\langle(X_E,X_R,X_I,X_G)\rangle. \\ ((\!|\mathsf{U}_{\mathrm{S}}(X_E,X_R,X_I,X_G)|\!) \,||\, \overline{m}\langle \mathsf{U}_{\mathrm{S}}(X_E,X_R,X_I,X_G)\rangle.\mathbf{0}))). \end{gathered}$$

The other input was blocked from synchronization with the output by the entailment relation.

Definition 2.4.18 (syntactic restrictions for dcrPsi): We define a dcrPsi-process P to be *syntactically correct* for a set of events E if it is of the following form, up to structural congruence:

$$P = (\nu m)((\prod_{0\leq k\leq g\leq k'} (\!|\Psi_g|\!)) \,||\, \overline{m}\langle(Ex_{k'}, Re_{k'}, In_{k'}, k')\rangle.\mathbf{0} \,||\, (\prod_{e\in E} P_e))$$

with $k \in \mathbb{N}$ and

$$\begin{gathered} \Psi_g := (\!|(Ex_g, Re_g, In_g, g)|\!) \quad \text{where } |Ex_g| = g, \\ P_e := !(\mathbf{case}\ \varphi_e : \underline{m}\langle(X_E,X_R,X_I,X_G)\rangle.(\overline{m}\langle \mathsf{U}_e(X_E,X_R,X_i,X_G)\rangle.\mathbf{0} \,||\, (\!|\mathsf{U}_e(X_E,X_R,X_i,X_G)|\!))\,)\ , \end{gathered}$$

with

$$\varphi_e = (\rightarrow\!\!\bullet e, \rightarrow\!\!\diamond e, e),$$

where the indexed Ex are multi-sets of events, and the indexed Re, In, as well as $\rightarrow\!\!\bullet e, \rightarrow\!\!\diamond e$ are sets of events.

The lemma below states some easy observations that we will use in the following.

Lemma 2.4.19: In a syntactically correct dcrPsi-process P we have that:

1. Different assertion processes have different generations, not necessarily starting at 0, and less or equal to the assertion in the unique output at the top level.
2. Each P_e sub-process corresponds to a unique event.
3. There is a unique sub-process $\overline{m}\langle(Ex, Re, In, g)\rangle.\mathbf{0}$ at top level, and moreover, $\mathcal{F}(P) = (Ex, Re, In, g)$.
4. The process P needs only five names $\{m, X_E, X_R, X_I, X_G\}$ since in each P_e the names $\{X_E, X_R, X_I, X_G\}$ are bound by the input construct.

In the following when we refer to a dcrPsi-process we will assume that it is a syntactically correct process, over some finite set E of event constants. In Theorem 2.4.28 we prove that for any syntactically correct P there exists a DCR-graph which is bisimilar to P. Before that we need a few preparatory results.

Lemma 2.4.20: For any $\mathcal{D}$ and Hi, the dcrPsi-process $\textsc{dcrpsi}(\mathcal{D}, Hi)$, if defined, is syntactically correct.

Proof: Follows directly from Definition 2.4.15 of $\textsc{dcrpsi}$. □

Lemma 2.4.21 (frame–marking correspondence): For any $\mathcal{D}$ and Hi, then the following statements are equivalent

- the frame of $\textsc{dcrpsi}(\mathcal{D}, Hi)$ is $(Hi, Re, In, |Hi|)$
- the marking of $\mathcal{D}$ is $(\lfloor Hi \rfloor, Re, In)$.

Proof: The $\textsc{dcrpsi}(\mathcal{D}, Hi)$ returns a dcrPsi process with only one assertion $(Hi, Re, In, |Hi|)$ which forms the frame. This assertion is made directly from the marking (Ex, Re, In) of $\mathcal{D}$, together with the given generation $|Hi|$, and we know that $\lfloor Hi \rfloor = Ex$. □

Lemma 2.4.22 (transitions preserve syntactic correctness): For any syntactically correct dcrPsi-process

$$P_0 \equiv (\nu m)((\prod_{k \le g \le g_0} (\!|\Psi_g|\!)) \,||\, \overline{m}\langle (Ex_0, Re_0, In_0, g_0)\rangle.\mathbf{0} \,||\, (\prod_{e\in E} P_e))$$

if $\mathbf{1} \rhd P_0 \xrightarrow{\alpha} P_1$ and $\mathcal{F}(P_i) = (Ex_i, Re_i, In_i, g_i)$, for $i \in \{0, 1\}$, then

1. $\alpha = \tau$
2. $$P_1 \equiv (\nu m)((\prod_{k \le g \le g_0+1} (\!|\Psi_g|\!)) \,||\, \overline{m}\langle (Ex_1, Re_1, In_1, g_1)\rangle.\mathbf{0} \,||\, (\prod_{e\in E} P_e))$$
3. $\exists e \in E$ such that
$$Ex_1 = Ex_0 + \{e\}, Re_1 = (Re_0 \setminus \{e\}) \cup e{\bullet\!\!\to}, In_1 = (In_0 \setminus e{\to\!\%}) \cup e{\to\!\!+}, g_1 = s(g_0)$$

in particular, it follows that transitions of syntactically correct dcrPsi-processes preserve syntactic correctness.

Proof:

1. In Psi-calculi there are three different types of transitions: input, output, and τ transitions. Syntactically correct dcrPsi-processes communicate over only one channel name m which is guarded, and therefore there cannot exist input nor output transitions. Thus the only possibility is τ-transitions.
2. To see that P_1 is syntactically correct note that any transition in P_0 is between the unique top level output $\overline{m}\langle (Ex_0, Re_0, In_0, g_0)\rangle.\mathbf{0}$ and a **case**-guarded input
$$\textbf{case } \varphi_e : \underline{m}\langle (X_E, X_R, X_I, X_G)\rangle.(\overline{m}\langle \mathsf{U}_e(X_E, X_R, X_i, X_G)\rangle.\mathbf{0} \,||\, (\!|\mathsf{U}_e(X_E, X_R, X_i, X_G)|\!))$$
coming from the replication of some P_e. After the transition and substitution updating the marking, the output becomes $\mathbf{0}$, whereas the **case** process becomes $\overline{m}\langle (Ex_1, Re_1, In_1, g_1)\rangle.\mathbf{0} \,||\, (\!|(Ex_1, Re_1, In_1, g_1)|\!)$ where Ex_1, Re_1, In_1, g_1 are as in the statement, cf. Definition 2.4.15 and Notation 2.4.16. Since $Ex_1 = Ex_0 + \{e\}$ we can find the event e responsible for the current τ-transition through $Ex_1 \setminus Ex_0 = \{e\}$. In consequence:
   (a) The single output of P_0 has been reduced in P_1 to $\mathbf{0}$ and we obtained exactly one new output of the correct form, as required.
   (b) Since no assertions from P_0 were removed, we can write the assertions in P_1 as $\prod_{k \le g \le g_0} (\!|\Psi_g|\!) \,||\, (\!|(Ex_1, Re_1, In_1, s(g_0))|\!)$, which can be written as $\prod_{k \le g \le s(g_0)} (\!|\Psi_g|\!)$.

(c) The application of rule (REP) reduces P_e to itself in parallel with the **case**-process that participated in the communication above. Therefore, the product of P_e processes remains the same, whereas the case process becomes the new output and a new assertion process.

3. Follows directly from the Notation 2.4.16 for U_e after substitutions.

□

that throughout this section we work with operations on multi-sets, but use the notation for sets, and rely on the context to disambiguate.

Based on Lemma 2.4.22 we can define transitions labelled by event constants between two dcrPsi-processes as follows.

Definition 2.4.23 (event transitions): Define $P \overset{e}{\rightsquigarrow} P'$ iff $\mathbf{1} \rhd P \xrightarrow{\tau} P'$ and $Ex' \backslash Ex = \{e\}$ from $\mathcal{F}(P) = (Ex, Re, In, g)$ and $\mathcal{F}(P') = (Ex', Re', In', g')$. Define *event-labelled transition systems*, denoted $\mathcal{LTS}$, to have as states all dcrPsi-processes and transitions between states defined by the above event-labelled transitions. For some dcrPsi-process P, we call the event-labelled transition system of P, denoted $\mathcal{LTS}(P)$, only the part of $\mathcal{LTS}$ reachable from P.

We are now ready to show that the dcrPsi-mapping preserves the behaviour.

Proposition 2.4.24 (preserving behaviour): For

$$P \equiv (\prod_{k \leq g < |Hi|} (\!|\Psi_g|\!)) \mid\mid \textsc{dcrpsi}(\mathcal{D}, Hi)$$

then

$$P \overset{e}{\rightsquigarrow} P' \iff \mathcal{D} \xrightarrow{e} \mathcal{D}'$$

s.t.

$$P' \equiv (\prod_{k \leq g < |Hi|} (\!|\Psi_g|\!)) \mid\mid (\!|\mathcal{F}(\textsc{dcrpsi}(\mathcal{D}, Hi))|\!) \mid\mid \textsc{dcrpsi}(\mathcal{D}', Hi + \{e\}).$$

Proof: First it is easy to see that $\mathcal{F}(P) = \mathcal{F}(\textsc{dcrpsi}(\mathcal{D}, Hi))$ from Definitions 2.4.15 and 2.4.18.

From Definitions 2.4.2, 2.4.9 and 2.4.15, and Lemma 2.4.21, we can see that an event is enabled in P iff it is enabled in $\mathcal{D}$. Therefore, whenever a transition exists on one side of the double implication, it exists on the other side as well.

It is easy to see from Definitions 2.4.3 and 2.4.15, that after a transition $P \overset{e}{\rightsquigarrow} P'$, the updates to the frame $\mathcal{F}(P) = (Ex, Re, In, g)$ that we see in the frame of $\mathcal{F}(P')$ are the same as when updating the marking to $M' = (Ex', Re', In')$ in the transition $\mathcal{D} \xrightarrow{e} \mathcal{D}'$, with the exception of Ex being a multi-set and the generation g. In particular, $Ex' = Ex + \{e\}$, which in a marking it means that e is added to Ex if it was not already there, whereas in the assertion the multiplicity of e is increased.

Therefore, P' is the same as P with the addition of the above new assertion process $(\!|\Psi_{|Hi+\{e\}|}|\!)$ left behind by the communication, and that the output has been changed to match the new state. Since this has highest generation, it becomes the frame of P'. Moreover, this assertion together with $\textsc{dcrpsi}(\mathcal{D}, Hi)$ are the same as $\textsc{dcrpsi}(\mathcal{D}', Hi + \{e\})$. □

Definition 2.4.25 (event bisimulation): We define event bisimulation $\overset{\bullet}{\sim}$ between dcrPsi-processes $P \overset{\bullet}{\sim} Q$, to be the standard bisimulation between their corresponding event-labelled transition systems $\mathcal{LTS}(P) \sim \mathcal{LTS}(Q)$. This particularly means that there exists a relation $R \subseteq \mathcal{LTS}(P) \times \mathcal{LTS}(Q)$ between the states of the two transition systems such that $(P, Q) \in R$ and for any e

1. if $P \overset{e}{\rightsquigarrow} P'$ then $\exists Q' \in$ s.t. $Q \overset{e}{\rightsquigarrow} Q'$ and $(P', Q') \in R$;
2. if $Q \overset{e}{\rightsquigarrow} Q'$ then $\exists P' \in$ s.t. $P \overset{e}{\rightsquigarrow} P'$ and $(P', Q') \in R$.

The same notion of event bisimulation can be defined for DCR-graphs over their event-labelled transition systems.

Theorem 2.4.26: For a DCR graph $\mathcal{D}$ with marking $M = (Ex, Re, In)$ then

$$\mathcal{LTS}(\mathcal{D}) \overset{\bullet}{\sim} \mathcal{LTS}(\text{DCRPSI}(\mathcal{D}, Ex)).$$

Proof: We denote by $\overrightarrow{E}$ a sequence of events, and by $\xrightarrow{\overrightarrow{E}}$ a sequence of transitions labelled by the respective evens in the sequence $\overrightarrow{E}$. Indexes are used to refer to elements in such sequences.

We show that the following set is a bisimulation:

$$\{(\mathcal{D}_k, P_k) \mid \mathcal{D} \xrightarrow{\overrightarrow{E}} \mathcal{D}_k, \text{DCRPSI}(\mathcal{D}, Ex) \overset{\overrightarrow{E}}{\rightsquigarrow} P_k\}.$$

This is equivalent to the single steps coinductive statement of Definition 2.4.25. The initial pair, for the empty sequence, is $(\mathcal{D}, \text{DCRPSI}(\mathcal{D}, Ex))$.

Proving that all pairs respect the requirements of Definition 2.4.25 is easy by induction over the length k. The induction has as basis the above initial pair. The Proposition 2.4.24 ensures that whenever a transition exists from one element of the pair, then the same transition exists from the other element. By the above construction, this new pair is in our bisimulation relation, thus respecting Definition 2.4.25. □

Proposition 2.4.27 (determinism in dcrPsi): Syntactically correct dcrPsi-processes are deterministic; i.e., in the execution graph, at any point P if $P \overset{e}{\rightsquigarrow} P'$ then P' is unique.

Proof: For any syntactically correct dcrPsi-processes we know that there is only one output process, cf. Lemma 2.4.19(3), and all input processes have a unique event e, cf. Lemma 2.4.19(2). Since event-labelled transitions are communications between one input and the single output, cf. Definition 2.4.23, then there can be at most one transition labelled by e. Lemma 2.4.22 ensures that any point in the execution graph is a syntactically correct process, thus finishing the proof. □

Theorem 2.4.28 (syntactic restrictions): For any syntactically correct dcrPsi-process P_{DCR}, i.e., built according to the syntactic restrictions in Definition 2.4.18, there exists a DCR-graph $\mathcal{D}$ s.t.

$$\mathcal{LTS}(\mathcal{D}) \overset{\bullet}{\sim} \mathcal{LTS}(P_{DCR}).$$

Proof: We take the set of events of the DCR-graph to be the set of event constants of the dcrPsi instance. We know from Definition 2.4.18 that P_{DCR} is built as

$$P = (\nu m)((\prod_{k \leq g \leq k'} (\!|\Psi_g|\!)) \,||\, \overline{m}\langle(Ex_{k'}, Re_{k'}, In_{k'}, k')\rangle.\mathbf{0} \,||\, (\prod_{e \in E} P_e))$$

where P_e is of the form

$$!(\mathbf{case}\ \varphi : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.(\overline{m}\langle \mathsf{U}_e(X_E, X_R, X_I, X_G)\rangle.\mathbf{0} \,||\, (\!|\mathsf{U}_e(X_E, X_R, X_I, X_G)|\!)\,)\,)$$

with $\varphi = (\rightarrow\!\bullet e, \rightarrow\!\diamond e, e)$ and $\mathsf{U}_e(X_E, X_R, X_I, X_G) = (X_E \cup \{e\}, (X_R \setminus \{e\}) \cup e\bullet\!\!\rightarrow, (X_I \setminus e\rightarrow\!\%) \cup e\rightarrow\!+, s(X_G))$ for $\rightarrow\!\bullet e$, $\rightarrow\!\diamond e$, $e\bullet\!\!\rightarrow$, $e\rightarrow\!\%$, $e \rightarrow\!+$ some subsets of E. We define the relations of the DCR-graph $\mathcal{D}$ from these subsets, i.e. for $e, e' \in E$ define $e' \rightarrow\!\bullet\, e$ if $e' \in \rightarrow\!\bullet e$, and similarly for the other relations.

Finally, define the marking M of $\mathcal{D}$ by taking the frame of the process P_{DCR}.

The bisimulation follows easily from Theorem 2.4.26. □

2.4.3 Correlation between dcrPsi and eventPsi

An interesting problem is to look more closely at the encoding of event structures through the ESPSI and the encoding through DCRPSI when seen as a special case of DCR-graphs; a question on these lines could be:

are ESPSI($\mathcal{E}$) and DCRPSI($\mathbf{dcr}(\mathcal{E})$) similar in any way?

Answering this question is not immediate. First of all, ESPSI translates into the eventPsi instance, whereas DCRPSI into the dcrPsi instance, and these two instances work with different terms and operator definitions. Even more, the encoding of event structures exhibits behaviour through *labelled transitions*, whereas for the encoding of DCRs we need to look into the frames of the processes before and after a transition to find the event that made this transition (cf. Lemma 2.4.22). Therefore, to find the correspondence that we are looking for we need to work in the same psi-instance, or establish correlations between the instances.

Let us first see how the DCRPSI($\mathbf{dcr}(\mathcal{E})$) process looks like, and then we will see a correlation between this and an eventPsi-process for the same $\mathcal{E}$.

Lemma 2.4.29: For an event structure $\mathcal{E}$ the dcrPsi-process DCRPSI($\mathbf{dcr}(\mathcal{E})$) has:

1. all conditions of the form $(Co, \emptyset, e)$ (for arbitrary Co and e) and
2. the initial assertion $(\emptyset, \emptyset, E, 0)$.

Proof: From Definition 2.4.5 we know that the DCR-graph $\mathbf{dcr}(\mathcal{E})$ has $\rightarrow\!\!\diamond = \emptyset$, which implies that the encoding function DCRPSI produces conditions $(Co, \emptyset, e)$ with the second element always empty. The initial assertion is made directly from the initial marking in the start process $P_{|Hi|}$ in Definition 2.4.15, which in the case of $\mathbf{dcr}(\mathcal{E})$ is $(\emptyset, \emptyset, E)$, cf. Definition 2.4.5. □

Lemma 2.4.30: If DCRPSI($\mathbf{dcr}(\mathcal{E}), Ex) \rightsquigarrow^* P \overset{e}{\rightsquigarrow} P'$ and frame $\mathcal{F}(P) = (Ex, \emptyset, In, g)$, then the new frame is $\mathcal{F}(P') = (Ex + \{e\}, \emptyset, In \setminus (\sharp e \cup \{e\}), s(g))$.

Proof : The fact that the generation increases to $s(g)$ is easy to see from how the assertions are created on transitions. The proof of Lemma 2.4.22 shows that the first element of the updated frame will be $Ex' = Ex + \{e\}$. From Definition 2.4.5 of $\mathbf{dcr}(\mathcal{E})$ we have $\rightarrow\!\%= \sharp \cup \{(e,e) \mid e \in E\}$ which implies that $\rightarrow\!\% \; e = (\sharp e \cup \{e\})$. Since $\rightarrow\!+ = \emptyset$, we have that the update of the third element of an assertion, when event e happens, is $In' = (In \setminus e\!\rightarrow\!\%) \cup e\!\rightarrow\!+ = (In \setminus (\sharp e \cup \{e\})) \cup \emptyset = In \setminus (\sharp e \cup \{e\})$. From Definition 2.4.5 we also have $\bullet\!\!\rightarrow = \emptyset$, which implies that the second element of the new frame is $Re' = (\emptyset \setminus \{e\}) \cup \emptyset = \emptyset$. □

Lemma 2.4.31: For a dcrPsi-process DCRPSI($\mathbf{dcr}(\mathcal{E}), Ex) \rightsquigarrow^* P$, if $\mathcal{F}(P) = (Ex, Re, In, g)$ then

1. $Re = \emptyset$,
2. $Ex \cap In = \emptyset$,
3. $In = E \setminus (\cup_{e \in Ex}(\sharp e \cup \{e\}))$.

Proof: For each of the three points of the statement we use an inductive argument. For 1 the base case is given by Lemma 2.4.29(2) which shows that in the execution of the process initially we have $Re = \emptyset$. For the inductive case we use Lemma 2.4.30.

For 2 the base case is given by Lemma 2.4.29 which says that initially $Ex \cap In$ is the same as $\emptyset \cap E = \emptyset$. For the inductive case assume that the frame is $\mathcal{F}(P) = (Ex, \emptyset, In, g)$, with $Ex \cap In = \emptyset$ from the induction hypothesis. Consider a transition $P \overset{e}{\rightsquigarrow} P'$ and we show that the second claim holds for P'. Having the transition implies that the event e is enabled in the frame of P; i.e., Definition 2.4.23 implies a communication with a **case** process P_e for which the φ_e is enabled by $\mathcal{F}(P)$, which by Definition 2.4.9 implies that $e \in In$. From Lemma 2.4.30 we know that

$\mathcal{F}(P') = (Ex' = Ex + \{e\}, Re' = \emptyset, In' = In \setminus (\sharp e \cup \{e\}), s(g))$ which only adds $\{e\}$ to Ex' compared to Ex. Since $e \in In$ it means that $Ex' \cap In = \{e\}$. But for In' we remove e from In, thus we have that $Ex' \cap In' = \emptyset$.

For 3 the base case is given by Lemma 2.4.29 which says that initially $In = E$ and $Ex = \emptyset$, thus allowing for the equality $In = E \setminus (\cup_{e\in\emptyset})(\sharp e \cup \{e\})$. For the induction case we assume that we have a process P with frame $(Ex, \emptyset, In, g)$ where $In = E \setminus (\cup_{e\in Ex}(\sharp e \cup \{e\}))$. Consider a transition $P \overset{f}{\rightsquigarrow} P'$, implying by definition that $f \in In$, and the new frame $\mathcal{F}(P') = (Ex' = Ex + \{f\}, \emptyset, In', s(g))$ which has $In' = In \setminus (\sharp f \cup \{f\})$ by Lemma 2.4.30. Using the induction hypothesis, this is equal to $E \setminus (\cup_{e\in Ex}(\sharp e \cup \{e\})) \setminus (\sharp f \cup \{f\}) = E \setminus (\cup_{e\in Ex'}(\sharp e \cup \{e\}))$. □

From the lemmas above we can define an embedding $\mathbf{emb}^g$, parametrised by some natural number $g \in \mathbb{N}$, from the assertions in eventPsi to the assertions in dcrPsi as follows.

Definition 2.4.32 (correlations between assertions): For Ψ_C an assertion of eventPsi and $g \in \mathbb{N}$ a natural number define

$$\mathbf{emb}^g(\Psi_C) = (\Psi_C, \emptyset, (E \setminus \Psi_C) \setminus (\cup_{e\in\Psi_C} \sharp e), g).$$

We used the notation Ψ_C to remind that the assertion in eventPsi is a set of events corresponding to a configuration C in the event structure, cf. Lemma 2.3.14. The above definition of embedding is motivated by Lemma 2.4.31, in particular, having the second element of the assertion being $\emptyset$ comes from Lemma 2.4.31(2) and the shape of the third element is because of Lemma 2.4.31(3).

The opposite direction of the embedding from Definition 2.4.32 is obvious the projection on the first element of the dcrPsi assertion, keeping only the support set of the multi-set; denote this projection by $\mathbf{prj}(\Psi)$.

From Lemma 2.4.29 we can define a similar embedding of conditions, which we will also denote by $\mathbf{emb}^e$. We use these embeddings in the proofs and definitions later on.

Definition 2.4.33 (correlations between conditions): For some event constant e and condition $(S_<, S_\sharp)$ of eventPsi define

$$\mathbf{emb}^e((S_<, S_\sharp)) = (S_<, \emptyset, e).$$

We can also define an opposite embedding $\overleftarrow{\mathbf{emb}}_S$, parametrised by a set of events S, taking a condition $(Co, \emptyset, e)$ of dcrPsi and returning a condition in eventPsi as follows:

$$\overleftarrow{\mathbf{emb}}_S((Co, \emptyset, e)) = (Co, S).$$

Lemma 2.4.34 (correlation between entailment relations): For an event structure $\mathcal{E}$ and its encoding $\textsc{espsi}(\mathcal{E})$, and one of its conditions $\varphi_e = (<e, \sharp e)$ entailed by some assertion Ψ, then

$$\Psi \vdash \varphi_e \quad \Rightarrow \quad \mathbf{emb}^g(\Psi) \vdash \mathbf{emb}^e(\varphi_e)$$

in dcrPsi, under the assumption that $e \notin \Psi$, and for some arbitrary $g \in \mathbb{N}$.

Proof: We need to prove that

$$\mathbf{emb}^g(\Psi) = (\Psi, \emptyset, (E \setminus \Psi) \setminus (\cup_{e'\in\Psi} \sharp e'), g) \vdash (<e, \emptyset, e) = \mathbf{emb}^e(\varphi_e)$$

under the assumptions that $<e \subseteq \Psi$ and $\sharp e \cap \Psi = \emptyset$, coming from the definition of entailment for eventPsi. For proving the above, the definition of entailment for dcrPsi requires that we prove three points (see Definition 2.4.9)

1. $e \in In = (E \setminus \Psi) \setminus (\cup_{e'\in\Psi} \sharp e')$;
2. $In \cap <e \subseteq \Psi$;
3. $(In \cap \emptyset) \cap \emptyset = \emptyset$;

(after obvious simplifications coming from the fact that we have special assertions and conditions, i.e., containing empty sets). It is easy to see that the second point holds because we already have that $<e \subseteq \Psi$. The third point is trivial since the sets of milestones and responses are empty in our case.

For the first point, the assumption that $e \notin \Psi$ implies that $e \in E \setminus \Psi$. To finish the proof it means we are left with proving the fact $e \notin \cup_{e' \in \Psi}\{e'' \in E \setminus \Psi_C \mid e' \in \sharp e''\}$, thus implying that $e \in In$. We do this proof by *reductio ad absurdum* and assume the contrary, i.e., $\exists e' \in \Psi : e \in \sharp e'$. We know from the fact that in event structures the conflict relation is symmetric, that if $e \in \sharp e'$ then also $e' \in \sharp e$. This, together with the fact that $e' \in \Psi$, implies that $\sharp e \cap \Psi \neq \emptyset$, which is a contradiction, thus making our assumption false and finishing the proof. □

In a dcrPsi-processes generated by the encoding function DCRPSI any event (i.e., the associated **case** processes P_e) may happen (i.e., participate in a communication) infinitely many times, as long as it is enabled. On the other hand, in an event structure an event can happen at most once.

Lemma 2.4.35: No event in a process $\text{DCRPSI}(\mathbf{dcr}(\mathcal{E}))$ can happen more than once.

Proof: For an event to be enabled the corresponding condition must be enabled by the assertion and the frame of the process, which from the definition implies that $e \in In$. We also know that initially there are no events that have happened and $Ex = \emptyset$. From Lemma 2.4.30 we know that after a transition the respective event is added to the new Ex and from Lemma 2.4.31 we know that any events in Ex cannot simultaneously be in In. Since Ex does not decrease we thus have that if an event e has happened it will never be enabled again. □

We are now ready to define an embedding of eventPsi processes in the context of Theorem 2.3.19, i.e., generated by ESPSI, into dcrPsi-processes. Assertions in eventPsi are sets and in the next definition we use the number of elements of the set Ψ as the g parameter that the $\mathbf{emb}^g$ requires when applied over assertions.

Definition 2.4.36 (embedding eventPsi into dcrPsi): We define an embedding function **emb** which takes an eventPsi-processes $\text{ESPSI}(\mathcal{E})$ generated from an event structure $\mathcal{E}$, which according to Definition 2.3.9 is $\text{ESPSI}(\mathcal{E}) = ||_{e \in E} P_e$ with

$$P_e = \begin{cases} (\!|\{e\}|\!) & \text{if } e \in C \\ \mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!) & \text{otherwise} \end{cases}$$

where $\varphi_e = (<e, \sharp e)$; and returns the dcrPsi-process $\mathbf{emb}(\text{ESPSI}(\mathcal{E}))$ as follows:

$$\begin{aligned} \mathbf{emb}(\text{ESPSI}(\mathcal{E})) \quad := \quad & (\nu m)(\mathbf{emb}(\,||_{e \in C}(\!|\{e\}|\!)) \,|| \\ & \overline{m}\langle \mathbf{emb}((\!|\otimes_{e \in C}\{e\}|\!))\rangle.\mathbf{0} \,|| \\ & ||_{P_e} \mathbf{emb}(P_e)) \end{aligned}$$

where $\mathbf{emb}((\!|\Psi|\!)) := (\!|\mathbf{emb}^g(\Psi)|\!)$, with $g = |\Psi|$, and P_e ranges over all the **case** processes from $\text{ESPSI}(\mathcal{E})$ with $\mathbf{emb}(P_e)$ being defined as:

$$\begin{aligned} \mathbf{emb}(P_e) \quad := \quad & !(\ \mathbf{case}\ \mathbf{emb}^e(\varphi_e) : \underline{m}\langle (X_E, X_R, X_I, X_G)\rangle. \\ & (\ \overline{m}\langle \mathsf{U}_e(X_E, X_R, X_I, X_G)\rangle.\mathbf{0} \,||\, (\!|\mathsf{U}_e(X_E, X_R, X_I, X_G)|\!)\ )\ ) \end{aligned}$$

with $\mathsf{U}_e(X_E, X_R, X_I, X_G) = (X_E \cup \{e\}, (X_R \setminus \{e\}) \cup \emptyset, (X_I \setminus (\{e\} \cup \sharp e) \cup \emptyset, s(X_G))$. When $C = \emptyset$, the empty composition becomes the minimal assertion $\mathbf{1}$.

Theorem 2.4.37: For an event structure $\mathcal{E}$ and an initial empty configuration $C = \emptyset$, we have:

$$\mathbf{emb}(\text{ESPSI}(\mathcal{E})) = \text{DCRPSI}(\mathbf{dcr}(\mathcal{E}), \emptyset).$$

Proof : On the left side of the equality the process $\text{ESPSI}(\mathcal{E})$ looks like $||_{e \in E}\mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)$, without any assertion process because the initial configuration is $C = \emptyset$. This process is embedded into dcrPsi, using Definition 2.4.36, as:

$$(\nu m)(\mathbf{emb}^0(\mathbf{1}) \,||\, \overline{m}\langle \mathbf{emb}^0(\mathbf{1})\rangle.\mathbf{0} \,||_{e \in E} \mathbf{emb}(P_e)).$$

Since the assertion $\mathbf{1} = (\emptyset, \emptyset, \emptyset, 0)$ then $\mathbf{emb}^0(\mathbf{1}) \,||\, \overline{m}\langle \mathbf{emb}^0(\mathbf{1})\rangle.\mathbf{0}$ becomes $(\!|\emptyset, \emptyset, E, 0|\!) \,||\, \overline{m}\langle(\emptyset, \emptyset, E, 0)\rangle.\mathbf{0}$. Each of the P_e are translated as in Definition 2.4.36.

On the right side of the equality, the Definition 2.4.5 for **dcr** says that the DCR $\mathbf{dcr}(\mathcal{E}) = (E, M, <, \emptyset, \emptyset, \emptyset, \sharp \cup \{(e,e) \mid e \in E\})$ has the initial marking $M = (\emptyset, \emptyset, E)$. This means that the P_k, $k = 0$, generated by DCRPSI$(\mathbf{dcr}(\mathcal{E}), 0)$ is the same as what we had on the left side above, i.e., $(\!|\emptyset, \emptyset, E, 0|\!) \,||\, \overline{m}\langle(\emptyset, \emptyset, E, 0)\rangle.\mathbf{0}$.

It remains to check that the processes P_e that Definition 2.4.15 of DCRPSI generates are the same as $\mathbf{emb}(P_e)$, when considering the empty sets that **dcr** generates. Recall that the P_e generated by DCRPSI, on the right of the equality, are:

$$!(\mathbf{case}\ \varphi_e : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.(\overline{m}\langle \mathsf{U}_e(X_E, X_R, X_I, X_G)\rangle.\mathbf{0} \,||\, (\!|\mathsf{U}_e(X_E, X_R, X_I, X_G)|\!)))\ ,$$

with $\varphi_e = (\rightarrow\!\!\bullet e, \rightarrow\!\!\diamond e, e)$ and

$$\mathsf{U}_e(X_E, X_R, X_I, X_G) = (X_E \cup \{e\}, (X_R \setminus \{e\}) \cup e\bullet\!\!\rightarrow, (X_I \setminus e\rightarrow\!\!\%) \cup e\rightarrow\!\!+, s(X_G)).$$

Since the response, milestone, and include relations are empty, then the above has $\varphi_e = (\rightarrow\!\!\bullet e, \emptyset, e)$ and

$$\mathsf{U}_e(X_E, X_R, X_I, X_G) = (X_E \cup \{e\}, (X_R \setminus \{e\}) \cup \emptyset, (X_I \setminus e\rightarrow\!\!\%) \cup \emptyset, s(X_G))$$

where $e\rightarrow\!\!\%$ is the same as $\{e\} \cup \sharp e$. This shows it is equal with $\mathbf{emb}(P_e)$ since $\mathbf{emb}^e(\varphi_e)$ is, by Definition 2.4.33, the same as $(<e, \emptyset, e)$, and $\rightarrow\!\!\bullet = <$ in our case. □

We must make sure that the embedding from Definition 2.4.36 is correct in the sense that it preserves behaviour, as expressed below.

Proposition 2.4.38 (embedding preserves behaviour): For an event structure $\mathcal{E}$, then ESPSI$(\mathcal{E})$ and $\mathbf{emb}(\text{ESPSI}(\mathcal{E}))$ have the same behaviour, i.e.:

$$\mathcal{LTS}(\text{ESPSI}(\mathcal{E})) \overset{\bullet}{\sim} \mathcal{LTS}(\mathbf{emb}(\text{ESPSI}(\mathcal{E}))).$$

Proof: In short, the proof follows from Lemma 2.4.34 that correlates the entailment relations, thus showing that whenever an event is enabled in the eventPsi-process then it is also enabled in its embedding.

Construct the bisimulation relation $\overset{\bullet}{\sim}$ between the states of the two event-labelled transition systems as follows; and then show that this respects the requirements of being a bisimulation. We take ESPSI$(\mathcal{E}) \overset{\bullet}{\sim} \mathbf{emb}(\text{ESPSI}(\mathcal{E}))$ to be the initial points. We continue the construction procedure exactly as in the proof of Theorem 2.4.26. Therefore, for any E' reachable from ESPSI$(\mathcal{E})$ by some event e, take the process P' reachable from $\mathbf{emb}(\text{ESPSI}(\mathcal{E}))$ by the same e, if one such exists, and put $E' \overset{\bullet}{\sim} P'$. The E' and P' are necessarily unique. In the proof all we are interested in is the structure of building this relation, which is given by the above procedure.

We are left with proving that for any pair E, P in the bisimulation, we have the following two statements

1. if $E \overset{e}{\leadsto} E'$ then there exists the transition $P \overset{e}{\leadsto} P'$
2. if $P \overset{e}{\leadsto} P'$ then there exists the transition $E \overset{e}{\leadsto} E'$

These are enough since the determinism and uniqueness of the construction of the relation $\overset{\bullet}{\sim}$ ensure that the reachable states E' and P' are bisimilar (for both statements), thus completing the requirements of Definition 2.4.25.

For the first statement we use Lemma 2.4.34.

For the second statement we need to prove $\mathbf{emb}^g(\Psi) \vdash \mathbf{emb}^e(\varphi_e) \implies \Psi \vdash \varphi_e$. This is the same as proving $\mathbf{prj}(\mathbf{emb}^g(\Psi)) = \Psi$ and $\overleftarrow{\mathbf{emb}}_{\sharp e}(\mathbf{emb}^e(\varphi_e)) = \varphi_e$. These are both easy to see from the definition of the embedding functions, and we have the second statement. □

2.5 Conclusions and outlook

We have presented encodings of the declarative event-based models for concurrency of finite prime event structures and finite DCR-graphs, a generalisation of event structures allowing finite representations of infinite computations, into corresponding instances of Psi-calculi. We proved that computation in the event structures and DCR-graph models corresponds to reduction steps in the corresponding Psi-processes. Moreover, for both encodings we made use of the expressive logic that Psi-calculus provides to capture the causality and conflict relations of the prime event structures and DCR-graphs. This made it possible to prove that action refinement is respected by the encoding of event structures.

For the encoding of DCR-graphs we made use of the communication mechanism of Psi-calculi, whereas for prime event structures this was not needed.

For both encodings we gave the syntactic restrictions that capture the Psi-processes that correspond precisely to prime event structures respectively DCR-graphs.

Finally, we proved that the encoding of DCR-graphs conservatively generalises the encoding of event structures. For this we showed that the two Psi-processes obtained by (i) mapping an event structure first to DCR-graphs and then to the dcrPsi-calculus, respectively (ii) mapping the same event structure first to the eventPsi-calculi and then embed this in the dcrPsi-calculus, are event-labelled bisimilar.

The purpose of our investigations was to provide Psi-calculi models of event based, non-interleaving (or causal) concurrency models as a first step towards a study of adaptable, distributed and mobile computational artefacts. We believe that we succeeded showing that the Psi-calculi is indeed well suited for representing causal, non-interleaving models for concurrency.

Nevertheless, the encodings leave open issues. Firstly, a discrepancy remains between the interleaving semantics based on SOS rules of Psi-calculi, and the non-interleaving nature of the two models we considered. In Chapter 3 are we starting work to provide Psi-calculi with non-interleaving semantics, through providing it for Pi-calculus. We then go on in Chapter 4 to discuss how this work can then be extended to the full Psi-calculi. Secondly, it is not completely satisfactory that the behaviour of the dcrPsi-processes is observed by a somewhat intentionally constructed event-labelled transition semantics. An improvement could be to consider a more compositional definition of event structures and DCR-graphs as considered in [DHS15a]. Thirdly, it would be interesting to look into adding responses to Psi-calculi as introduced in DCR-graphs, allowing to represent liveness/progress properties, continuing along the lines of the work in [CHPW12] for Transition Systems with Responses.

The next steps towards studying adaptable, distributed and mobile computational artefacts will be to consider cases of workflows identified in field studies within the CompArt project and the notion of run-time refinement and adaptation supported by DCR-graphs as presented in [DHS15a]. By embedding DCR-graphs in the richer framework of Psi-calculi we anticipate being able to experiment with richer process models, e.g. representing locations, mobility and resources used by actors in workflows.

Remark 2.5.1 (on infinite set of events): If one would want to apply our encoding to infinite event structures then the set E may be infinite, hence the process and individual elements of $\mathbf{A}$ and $\mathbf{C}$ may be infinite terms (i.e., infinite sets). In the encoding produced by ESPSI, the conditions $\pi_L(\varphi_e)$ would be finite, because of the principle of finite causes of Definition 1.2.4 that event structures respects. Still, the $\pi_R(\varphi_e)$ may be infinite, because there is no restriction on the conflict relation in event structures, and thus an event can be in conflict with infinitely many events.

A simple example where this would appear is pictured in Figure 2.8, where we have a labelled transition system on the left and its unfolding as a labelled event structure on the right. The loop is unfolded into infinitely many sequential events, and for every second event we have a branch with a new c-labelled event which is in conflict with the rest of the infinite a, b labelled sequence. Executing one of these c-labelled events would mean cancelling all the infinitely many events that encode this branch. That is to say, the single event is in conflict with all the events on the looping branch, which are infinitely many.

Assertion terms from $\mathbf{A}$, produced by ESPSI, are always finite because they encode, cf. Lemma 2.3.14, configurations, which are finite sets. Still, it is problematic to have the infinite right part of the conditions, since it is used in deciding the entailment relation where one needs to decide if the intersection of an infinite and finite set is empty.

Besides this, the encoding ESPSI would result in infinitely many parallel processes if the set of events is infinite,

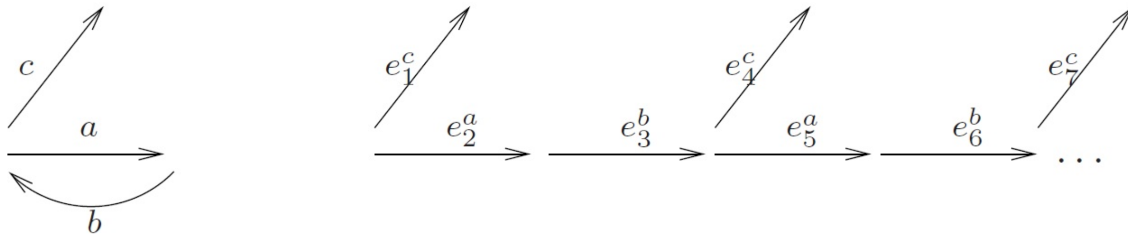


Fig. 2.8: Example of cancelling infinitely many different events coming from a loop unfolding. On the left we used a process graph view; on the right the event structures representation.

since a process is created for each $e \in E$. For practical use, infinite terms are not desirable, but for a theoretical encoding they could be fine, just like e.g. infinite summation in Milner's work on SCCS, infinite case construct for Psi-calculi, or infinite conjunctions in some logics.

When building infinite nominal terms and infinite Psi-processes one has to take care of not using infinitely many different free names, i.e., not to have infinite support for the nominal terms. Otherwise essential properties like alpha-renaming fail to work [GP02].

3. NON-INTERLEAVING SEMANTICS FOR PI-CALCULUS

We give in this chapter the first non-interleaving early operational semantics for the pi-calculus which generalizes the standard interleaving semantics and unfolds to the stable model of prime event structures.

Our starting point is the non-interleaving semantics given for CCS by Mukund and Nielsen, where the so-called *structural* (prefixing or subject) causality and events are defined from a notion of locations derived from the syntactic structure of the process terms. The semantics is conservatively extended with a notion of *extruder histories*, from which we infer the so-called *link* (name or object) causality and events introduced by the dynamic communication topology of the pi-calculus. We prove that the semantics generalises both the standard interleaving early semantics for the pi-calculus and the the non-interleaving semantics for CCS. In particular, it gives rise to a labelled asynchronous transition system unfolding to prime event structures.

3.1 Introduction

The pi-calculus [MPW92] is the seminal model for concurrent mobile processes, representing mobility by the fresh creation and communication of channel names. The standard operational semantics adopt an *interleaving* approach to concurrency, that represent concurrent execution of actions as their arbitrary sequential interleaving and employ basic transition systems or automata as semantic models. However, the ability to distinguish concurrency from interleaving has several practical applications, including dealing with state-space explosion in model-checking [CGMP99], supporting action refinement [vGG01] and reversibility (e.g. [UPY14, LMS16]).

To give a non-interleaving semantics one needs to identify the underlying events and their concurrency and causality relationships, and from that define a notion of non-interleaving observations, e.g. in terms of a bisimulation or testing equivalence [San96, vGG01] or employ a non-interleaving model (e.g. [NPW79, WN95, Bed88, Shi85b, BG95]), in which the concurrency can be represented explicitly. The dynamic communication topology of the pi-calculus makes it non-trivial to identify what accounts for causality, and indeed several possible approaches have been proposed. As described in [BS95], the source of the complexity is that the causal dependencies fall in two categories: The *structural* (prefixing or subject) dependencies, coming from the static process structure, i.e. action prefixing and parallel composition, and the *link* (name or object) dependencies, which come from the dynamic creation of communication links by scope extrusion of local names.

There has been quite some work on providing non-interleaving semantics for pi-calculus [MP95, JJ95, BG95, BS95, San96, CS00, CVY12, CKV13, Cri15] and process algebras in general (e.g. [BCHK94, Cas01]).

Among the most recent work, a stable operational semantics for reversible, deterministic and finite pi-calculus processes is provided in [CKV13]. Stability means that every event depends on a unique history of past events, which supports reversibility of computations. A denotational semantics for the pi-calculus is provided in [CVY12] as extended event structures. The semantics discards the property of stability to avoid the complexity and increase in number of events arising from achieving unique dependency histories. Both papers consider the late style pi-calculus semantics, where names received from the environment are kept abstract and thus distinct from any previously extruded names. We found no prior work providing a stable, non-interleaving, *early* style structural operational semantics generalising the standard early operational semantics of the pi-calculus. In the early style semantics, names received from the environment are concrete, and thus may be identical to a previously extruded name. Consequently, the choice between late and early style semantics influences the link causality.

Our key contribution is to provide the first stable, non-interleaving operational *early* semantics for the pi-calculus that generalises the standard, non-interleaving early operational semantics for the pi-calculus [SW01]. Our starting point is the work of Mukund and Nielsen [MN92], which defines a structural operational non-interleaving semantics for Milner's CCS [Mil80] as (labelled) asynchronous transition systems using locations to identify the structural

causality, which is the only type of causality in CCS. We generalize the approach of [MN92] to the pi-calculus by, in addition to the structural locations, employing a notion of *extruder histories*, recording the location of both name extrusions and name inputs. Together, the locations and extruder histories allow to identify the underlying events of transitions and both their structural and link causal dependencies.

Overview of chapter: In Sec. 3.2.1 we generalise the structural operational early semantics for the pi-calculus with locations for transitions, extrusion histories and link dependencies. In Sec. 3.4 we show that the semantics yields a standard labelled asynchronous transitions system, which is known to unfold to labelled prime event structures [WN95]. We conclude and comment on future work in Sec. 3.5. The work in this chapter is extended work of what was presented at LATA2017 [HJN17].

3.2 Causal Early Operational Semantics

In this section we give an early operational semantics of the pi-calculus recording both the structural and link causal dependencies between events.

We first recall the syntax for the pi-calculus with guarded choice.

3.2.1 Pi-calculus syntax

Definition 3.2.1: The set of *Pi-calculus processes* **Proc**, ranged over by P, Q, are defined using an infinite set of names $\mathcal{N}$, ranged over by small letters, by the grammar:

$$P ::= \Sigma_{i \in I} \varphi_i.P_i \mid (\nu n)P \mid P \,||\, Q \mid !P \mid \mathbf{0}, \qquad \varphi ::= \overline{a}\langle n \rangle \mid \underline{a}(n)$$

providing constructs for, respectively: guarded non-deterministic choice of output and input prefixes, restriction of name n, parallel composition, replication, and the empty/trivial process. For a process P we denote by $n(P)$ the set of *all names* appearing in P, by $bn(P)$ the *bound* names, i.e. those n that are restricted by (νn) or by the input action $\underline{a}(n)$, and by $fn(P) = n(P) \backslash bn(P)$ the *free* names. For an action label α we use the same notation $n(\alpha)$ for the similar notion. We assume all bound names are unique in a process and identify processes up to α-conversion. We often omit the indexing set I in the notation for a sum process $\Sigma_{i \in I} \varphi_i.P_i$ and use only $\Sigma \varphi_i.P_i$. When $I = \emptyset$ then we write $\mathbf{0}$ instead of $\Sigma_{i \in I} \varphi_i.P_i$, and when I is singleton we omit the sum symbol, e.g., $\overline{a}\langle n \rangle.P$.

For the particular process $(\nu n)\underline{a}(x).\mathbf{0}$ we have $n((\nu n)\underline{a}(x)) = \{n, a, x\}$, $bn((\nu n)\underline{a}(x)) = \{n, x\}$, and $fn((\nu n)\underline{a}(x)) = \{a\}$. We make a habit of using $a, b, ...$ for names that are meant to indicate channels/links for communication, $m, n, ...$ for names that are being transmitted over channels, and $x, y, z, ...$ when we intend these names to be substituted.

In Fig. 3.2 we give the causal early semantics for pi-processes with transitions of the form $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$. Following the approach in [MN92] we have added *location labels* u under the transitions, identifying the location of the prefixes in the term contributing to a transition and allowing to infer structural (CCS-like) events and causalities. To capture the finer notion of events and link causalities of the pi-calculus, we enrich the semantics with *extrusion histories* $H = (\overline{H}, \underline{H})$ to the left of the turnstile, which record the location in the parallel process of the prefixes extruding respectively receiving some name. (The precise definition is given later in Definition 3.2.11 after we prepare the ground for it.)

Example 3.2.2 (Non-interleaving vs. interleaving): If we have the processes "a then b or b then a" (written as $a.b + b.a$), and "a and in parallel b" (written as $a \,||\, b$), then it become impossible to see which one we get looking only at the execution trace. This can be seen in Figure 3.1 where we see the transition systems, that is a system with states and transitions between states, we get from the above two processes. An execution trace is any path we can take through a transition system.

Formally, we define the set of prefix locations as follows.

Definition 3.2.3: Let $\mathcal{L} = \{0, 1\}^* \times \mathbf{Proc} \times \mathbf{Proc}$ be the set of *prefix locations* and write $s[P][P']$ for elements in $\mathcal{L}$.

Proposition 3.2.4 (location labels): The location labels u of transitions in Fig. 3.2 are of the following forms:

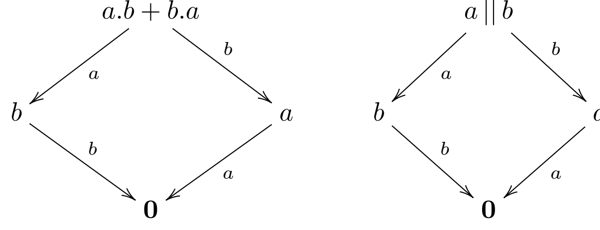


Fig. 3.1: Non-interleaving vs. interleaving diamond distinguished through different location labels.

1. $s[P][P'] \in \mathcal{L}$, if α is an input or output action,
2. $s\langle 0s_0[P_0][P'_0], 1s_1[P_1][P'_1]\rangle$, for $sis_i[P_i][P'_i] \in \mathcal{L}$ and $i \in \{0,1\}$, if $\alpha = \tau$.

Proof: By induction on the length of the derivation tree of the transition $H, P \xrightarrow[u]{\alpha} H', P'$. □

In words, a prefix location $s[P][P']$ provides a path $s \in \{0,1\}^*$ to an input/output prefixed subterm P through the abstract syntax tree, with 0 and 1 referring to the left respectively right component of a parallel composition, and P' being the residual sub-term after the transition. The location labels of the second form provide two prefix locations, $sis_i[P_i][P'_i] \in \mathcal{L}$ for $i \in \{0,1\}$, identifying the output and input prefix in a communication.

Note that to keep locations of action prefixes fixed, we cannot assume the usual structural congruence making parallel composition commutative. Therefore we must use two rules (PAR_i), $i \in \{0,1\}$ for parallel composition and two rules (COM_i), $i \in \{0,1\}$ for communication.

Example 3.2.5 (associativity and commutativity break location labels): Often in presentations of pi-calculus one considers structural rules like for the parallel composition being commutative and associative. We cannot have such structural rules because of how we create the location labels of the events and how we define the independence relation on events. For the commutativity of parallel composition we can easily see how our labels do not work any more, in the example process $P = \overline{a}\langle a\rangle \,||\, \overline{b}\langle b\rangle$. Here we can get the event $e = (\overline{a}\langle\epsilon\rangle a, 0[\overline{a}\langle a\rangle][\mathbf{0}])$ for the transition on channel a. If we had a structural rule saying that parallel composition can commute, i.e., that the two processes $P = \overline{a}\langle a\rangle \,||\, \overline{b}\langle b\rangle = \overline{b}\langle b\rangle \,||\, \overline{a}\langle a\rangle = P'$ are equivalent for the semantic rules, we would see $P' = \overline{b}\langle b\rangle \,||\, \overline{a}\langle a\rangle$ executing the same event for the same channel a but which has a different location label $e = (\overline{a}\langle\epsilon\rangle a, 1[\overline{a}\langle a\rangle][\mathbf{0}])$. This breaks our reasoning about which components execute which events. For the associativity of parallel consider the process $P' = (\overline{a}\langle a\rangle \,||\, \overline{b}\langle b\rangle) \,||\, \overline{c}\langle c\rangle$ where the event for b would be $(\overline{b}\langle\epsilon\rangle b, 01[\overline{b}\langle b\rangle][\mathbf{0}])$. If we allow the parallel to be associative we could write $P' = \overline{a}\langle a\rangle \,||\, (\overline{b}\langle b\rangle \,||\, \overline{c}\langle c\rangle)$ where the event for b would be $(\overline{b}\langle\epsilon\rangle b, 10[\overline{b}\langle b\rangle][\mathbf{0}]$. These two events again have completely different location labels, though they should be seen as comming from the same component.

From the locations we define our first notion of structural events and independence, which correspond to the events and independence defined for CCS in [MN92]. We will later show how to take into account the additional link causal relationships of the pi-calculus.

Definition 3.2.6 (structural events): Let $Ev = \{(\alpha, u) \mid (\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'\}$ be the *structural events*. For $e = (\alpha, u) \in Ev$ define $Loc(e) \subseteq \{0,1\}^*$, the *locations* where e occurs, by

$$Loc(e) = \begin{cases} \{s\} & \text{if } u = s[P][P'] \\ \{ss_0, ss_1\} & \text{if } u = s\langle s_0[P_0][P'_0], s_1[P_1][P'_1]\rangle. \end{cases}$$

Definition 3.2.7 (structural independence): Define an *independence relation on locations* $I_l \subseteq \{0,1\}^* \times \{0,1\}^*$ by

$$(s_0, s_1) \in I_l \quad \text{iff} \quad (s_0 = s0s'_0 \wedge s_1 = s1s'_1) \vee (s_0 = s1s'_0 \wedge s_1 = s0s'_1),$$

where $s, s_0, s_1, s'_0, s'_1 \in \{0,1\}^*$. Define the *structural independence* relation on events $I_{CCS} \subseteq Ev \times Ev$ by:

$$(e, e') \in I_{CCS} \quad \text{iff} \quad \forall s \in Loc(e), \forall s' \in Loc(e') : (s, s') \in I_l.$$

$$\frac{u = [\underline{a}(m).P][P'] \quad P' = P[m := n]}{(\overline{H}, \underline{H}) \vdash \underline{a}(m).P \xrightarrow[u]{\underline{a}(n)} (\overline{H}, \underline{H} \cup \{(n,u)\}) \vdash P'} \; (\text{IN})$$

$$\frac{\text{u=}[\overline{a}\langle n\rangle.\text{P}][\text{P}]}{(\overline{H}, \underline{H}) \vdash \overline{a}\langle n\rangle.P \xrightarrow[u]{\overline{a}\langle\epsilon\rangle n} (\overline{H}, \underline{H}) \vdash P} \; (\text{OUT})$$

$$\frac{(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\overline{a}\langle\epsilon\rangle n} (\overline{H}', \underline{H}') \vdash P' \qquad n \neq a}{(\overline{H}, \underline{H}) \vdash (\nu n)P \xrightarrow[u]{\overline{a}\langle n\rangle n} (\overline{H}' \cup \{(n,u)\}, \underline{H}') \vdash P'} \; (\text{OPEN})$$

$$\frac{(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P' \qquad b \notin n(\alpha)}{(\overline{H}, \underline{H}) \vdash (\nu b)P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash (\nu b)P'} \; (\text{SCOPE})$$

$$\frac{(\overline{H}, \underline{H}) \vdash P \,||\, !P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'}{(\overline{H}, \underline{H}) \vdash !P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'} \; (\text{REP})$$

$$\frac{(\overline{H}, \underline{H}) \vdash \varphi_i.P_i \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'}{(\overline{H}, \underline{H}) \vdash \Sigma_{i \in I} : \varphi_i.P_i \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'} \; (\text{SUM})$$

$$\frac{\begin{array}{cc} \overline{H}'' = \{(n,u) \mid \alpha = \overline{a}\langle\epsilon\rangle n, n \in \mathsf{dom}([j]\overline{H}), & P_j = P_j' \text{ and } j = 1-i \\ \forall l.l|_{\{0,1\}} \prec iu : (n,l) \notin \overline{H} \cup \underline{H}\} & \tilde{b} = \mathsf{dom}(\overline{H}_i')\backslash\mathsf{dom}([\check{i}]\overline{H}) \\ ([\check{i}]\overline{H}, [\check{i}]\underline{H}) \vdash P_i \xrightarrow[u]{\alpha} (\overline{H}_i', \underline{H}_i') \vdash P_i' & \tilde{b} \cap fn(P_j) = \emptyset \end{array}}{(\overline{H}, \underline{H}) \vdash P_0 \,||\, P_1 \xrightarrow[iu]{\alpha} ((\overline{H}\backslash[i]\overline{H}) \cup i\overline{H}_i' \cup i\overline{H}'', (\underline{H}\backslash[i]\underline{H}) \cup i\underline{H}_i') \vdash P_0' \,||\, P_1'} \; (\text{PAR}_i)$$

$$\frac{\begin{array}{cc} \underline{H}'' = \{(n, v_j) \mid & \tilde{b} = \mathsf{dom}(\overline{H}_i')\backslash\mathsf{dom}([\check{i}]\overline{H}) \\ \exists(n,l) \in [i](\overline{H} \cup \underline{H}) : l|_{\{0,1\}} \prec v_i\} & \tilde{b} \cap fn(P_j) = \emptyset, \quad j = 1-i \\ ([\check{i}]\overline{H}, [\check{i}]\underline{H}) \vdash P_i \xrightarrow[v_i]{\overline{a}\langle\tilde{b}\rangle n} (\overline{H}_i', \underline{H}_i') \vdash P_i' & ([\check{j}]\overline{H}, [\check{j}]\underline{H}) \vdash P_j \xrightarrow[v_j]{\underline{a}(n)} (\overline{H}_j', \underline{H}_j') \vdash P_j' \end{array}}{(\overline{H}, \underline{H}) \vdash P_0 \,||\, P_1 \xrightarrow[\langle 0v_0, 1v_1\rangle]{\tau} (\overline{H}, \underline{H} \cup j\underline{H}'') \vdash (\nu\tilde{b})(P_0' \,||\, P_1')} \; (\text{COM}_i)$$

Fig. 3.2: Early operational semantics enriched with *action labels* $\alpha ::= \tau \mid \overline{a}\langle\epsilon\rangle n \mid \underline{a}(x)$, *locations* u (under the arrows) and *extruder histories* $(\overline{H}, \underline{H})$ (to the left of the turnstile). We identify processes up-to α-equivalence, assume unique bound names, and that for all rules, if $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ is the conclusion, we require $\mathsf{dom}(\overline{H} \cup \underline{H}) \cap bn(P) = \emptyset$. For rules (COM) and (PAR$_i$), consider $i \in \{0,1\}$, and let $\tilde{b} = \mathsf{dom}(\overline{H}_i')\backslash\mathsf{dom}([\check{i}]\overline{H})$ and allow writing $(\nu\emptyset)P$ and $(\nu\{n\})P$ for P and $(\nu n)P$ respectively. The blue text shows what is added to the standard semantics. We elided a symmetric (COM) rule.

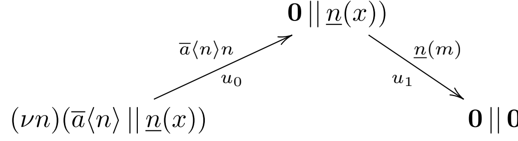


Fig. 3.3: Link causality exemplified in Ex.3.2.8, with $u_1 = 1[\underline{n}(x)][\mathbf{0}]$ and $u_0 = 0[\overline{a}\langle n\rangle][\mathbf{0}]$.

As the following example shows, I_{CCS} misses the link dependencies induced by extrusion of names.

Example 3.2.8 (link causality): Consider the process $(\nu n)(\overline{a}\langle n\rangle \,||\, \underline{n}(x))$, pictured in Figure 3.3. According to I_{CCS} the two events $e = (\overline{a}\langle n\rangle n, 0[\overline{a}\langle n\rangle][\mathbf{0}])$ and $e_m = (\underline{n}(m), 1[\underline{n}(x)][\mathbf{0}])$ are independent, for any $m \neq n$, but the semantics does not allow the input event to happen until after the name n has been extruded, i.e. there is an objective dependency between the extruding output and the input (this was observed in e.g. [CVY12, DP99]).

The next example shows that a name may have several parallel extruders, giving rise to a *disjunctive causality*. This shows that parallel extrudes introduce non-stability since prime event structures and asynchronous transition systems can only capture stable, or *conjunctive causality*. Disjunctive causality can sometimes be captured through splitting of events, which is the essence of what we do in Definition 3.2.12.

Example 3.2.9 (parallel extruders): Consider the process $(\nu n)(\overline{a}\langle n\rangle \,||\, (\overline{b}\langle n\rangle \,||\, \underline{n}(x)))$, which has two *parallel extruders* $\overline{a}\langle n\rangle$ and $\overline{b}\langle n\rangle$. We have the three events $e_a = (\overline{a}\langle n\rangle n, 0[\overline{a}\langle n\rangle][\mathbf{0}])$, $e_b = (\overline{b}\langle n\rangle n, 10[\overline{b}\langle n\rangle][\mathbf{0}])$, and $e_n = (\underline{n}(m), 11[\underline{n}(x)][\mathbf{0}])$. The event e_n for the input action is independent of both output events, but it cannot happen before at least one has happened.

The next example illustrates that both names in an input action $\underline{m}(n)$ may have been previously extruded, giving rise to a *conjunctive causality*.

Example 3.2.10 (two names in one action): Consider the process $P = (\nu n)(\nu m)(\overline{a}\langle n\rangle \,||\, (\overline{b}\langle m\rangle \,||\, \underline{m}(x)))$. From $(\emptyset, \emptyset) \vdash P$ we can have two extruding outputs on channels a and b of the names n respectively m, after which the output history would contain two pairs $\overline{H} = \{(n, 0[\overline{a}\langle n\rangle][\mathbf{0}]), (m, 10[\overline{b}\langle m\rangle][\mathbf{0}])\}$. We now may have an input action with label $\underline{m}(n)$ that depends on both extruder's.

The extrusion histories to the left of the turnstile helps us take into account the link causalites.

Definition 3.2.11 (extrusion histories): A *history* $H \subseteq \mathcal{H} = \mathcal{N} \times \mathcal{L}$ is a relation between names and prefix locations, and an *extrusion history* $(\overline{H}, \underline{H})$ is a pair of histories, referred to as the *output history* and *input history* respectively. For a history H and $i \in \{0, 1\}$, we use the following notations

- $iH = \{(n, iu) | (n, u) \in H\}$,
- $[\check{i}]H = \{(n, u) \mid (n, iu) \in H\}$,
- $[i]H = \{(n, iu) \mid (n, iu) \in H\}$,
- $[\check{\epsilon}]H = H$.

We may apply these notations several times and abbreviate them by a string, as in the example $[\check{0s}]H = [\check{s}][\check{0}]H$ with s a possibly empty location string. Finally, let $\mathsf{dom}(H) = \{n \mid (n, u) \in H\}$, i.e. the set of names recorded in the history.

Instead of a single event e_n in Example 3.2.9 we use the extrusion histories to split the event e_n in two conflicting events, e_{na} and e_{nb}, one causally dependent on output event e_a and one dependent on output event e_b. Crafa et al. in [CVY12, Sec.6] deems the approach of splitting events intractable and instead follows a different approach by defining a specially tailored event structure model incorporating a global set of extruded names. Our approach,

however, works with the standard stable model of asynchronous transition systems (which unfolds to prime event structures [WN95, Ch.10]) since we split events involved in disjunctive causalities using the insights from [CVY12].

Instead of the global set of extruded names, we use extruder histories to record, during the run of a process, in the output history $\overline{H}$ those names that have been extruded and at which location the output happened; whereas in the input history $\underline{H}$ we record names received on an input operation as well as at which location. Both sets are needed to decide the correct extruding events in the rules, and thus how to split events in Definition 3.2.12. The accumulation of extruded names is done in the (OPEN) and (PAR) rules, whereas the accumulation of received names is done in the (IN) rule. Extruder information are immediately available in the (OPEN) rule (which we call *initial extruder's*), but other events in parallel, under the same scope, may also be extruder's (which we call *possible extruder's*). The possible extruder's have the same relevance for splitting disjunctive causalities as the initial extruder's have. Therefore, they are collected in the same history element, which is done by the (PAR) rules.

Based on the examples above, we refine our transitions and events to also capture link dependencies by enriching the transitions with a set D, which we call deterministic sub-history, recording the link dependencies for each non-output name in α, i.e. the past extruding events a name depends on, if it was extruded in the past.

Definition 3.2.12 (causal semantics): Define the *causal* early semantics as the following transitions:
$(\overline{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ if $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ and

1. $D \subseteq \overline{H}$,
2. $(n, l), (n, l') \in D$ implies $l = l'$
3. $\mathsf{dom}(D) = \mathsf{dom}(\overline{H}) \cap no(\alpha)$,

where $no(\alpha)$ is the *non output names* of α, defined by $no(\overline{n}\langle x\rangle m) = \{n\}\backslash\{m\}$, $no(\underline{n}(m)) = \{n, m\}$ and $no(\tau) = \emptyset$.

The set D contains only one entry per name, even if several entries for the same name can be found in the history (coming from multiple extrusions). Moreover, D is the largest such set. In consequence, several D sets can be extracted from the same SOS transition, thus multiplying the number of causal transitions according to the different ways names could have been extruded. Note that the histories and processes are not changed by the above definition.

Proposition 3.2.13: The causal semantics is more fine grained than the structural semantics of Fig. 3.2 in the sense that:

1. if $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ then $\exists D \subset \mathcal{H}$ s.t. $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\overline{H}', \underline{H}') \vdash P'$;
2. if $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ then $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ was derived with the SOS rules of Fig. 3.2.

Proof: Trivial (note that D may be $\emptyset$). □

The link dependencies D allow us to define our final notion of events and independence relation for the causal semantics.

Definition 3.2.14 (causal independence): Let the set of *events* **Ev** be defined by:

$$\mathbf{Ev} = \{\big((\alpha, u), D\big) \in Ev \times \mathcal{H} \mid (\overline{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\overline{H}', \underline{H}') \vdash P'\}$$

Two events $e_i = (e_i', D_i) \in \mathbf{Ev}$ for $e_i' = (\alpha_i, u_i)$ and $i \in \{0, 1\}$ are independent, written $e_0 I e_1$, iff

$$e_0' I_{CCS}\, e_1' \ \wedge\ \nexists n : D_i(n) = u_{1-i} \quad \text{for } i \in \{0, 1\}.$$

Note that from the irreflexivity of the structural independence relation I_{CCS} , two events (e, D) and (e, D') that only differ on the splitting sets, i.e. $D \neq D'$, can never be independent. Returning to Example 3.2.9, the event $e_n = (\underline{n}(m), 11[\underline{n}(x)][\mathbf{0}])$ will be split in two events $\big(e_n, (n, 0[\overline{a}\langle n\rangle][\mathbf{0}])\big)$ and $\big(e_n, (n, 10[\overline{b}\langle n\rangle][\mathbf{0}])\big)$ corresponding to transitions between the same two states.

We now briefly explain the rules in Fig. 3.2.

The (IN) rule is the standard early input rule, substituting a received name n for the parameter m in P, yielding $P' = P[m := n]$, and enriched by recording the location $u = [\underline{a}(m).P][P']$ on the transition. Moreover, the rule takes care to add the name n to the input history $\underline{H}$.

The (OUT) rule is the standard output rule, enriched by bookkeeping the location $u = [\overline{a}\langle n\rangle.P][P']$ to the transition.

The (OPEN) rule is standard, except the location is recorded for the extruded name n in the output history and not in the action label, as is custom for the standard pi-semantics. Avoiding name extrusions in the labels ensures unique labels for events in Section 3.4, and only one (COM) rule. Otherwise, if we were to follow the pi-calculus conventions and have different action labels we would need to complicate the technicalities by adding a notion of equivalence of events (which would equate action labels with and without bound names), and proving results such as the independence relation has to respect this event equivalence (this more complicated approach can be seen in the technical report [NJH16a]).

The (SCOPE), (REP) and (SUM) rules are the standard rules, just extended to retain locations and histories.

If we do not consider the locations and histories, the (PAR_i) rules, for $i \in \{0, 1\}$, are the standard left and right parallel rules, except that we need to extract possibly extruded name from the histories by the set $\tilde{b} = \mathsf{dom}(\overline{H}'_i)\backslash\mathsf{dom}([\check{i}]\overline{H})$, and not from the action label α. The location label is extended by prefixing with $i \in \{0, 1\}$, to record in which component of the parallel composition the action happened. The extruder's recorded in the set $\overline{H}''$ capture exactly the parallel extrusion illustrated in Example 3.2.9. Specifically, an output location is added to the output extruder history, if the name has been extruded in the other parallel component and not previously extruded (recorded in the output history) nor received (recorded in the input history) by the current component. We illustrate the use of the input history in the Example 3.2.15.

Example 3.2.15: Consider the process $P = (\nu n)(\overline{a}\langle n\rangle \,||\, \underline{b}(x).\overline{c}\langle n\rangle)$. Starting with empty histories, we have the two transitions

1. $(\emptyset, \emptyset) \vdash P \xrightarrow[0[\overline{a}\langle n\rangle][\mathbf{0}]]{\overline{a}\langle n\rangle n} (\{(n, 0[\overline{a}\langle n\rangle][\mathbf{0}])\}, \emptyset) \vdash P_1$, for $P_1 = \mathbf{0} \,||\, \underline{b}(x).\overline{c}\langle n\rangle$

2. $(\emptyset, \emptyset) \vdash P \xrightarrow[1[\underline{b}(x).\overline{c}\langle n\rangle][\overline{c}\langle n\rangle]]{\underline{b}(m)} (\emptyset, \{(m, 1)\}) \vdash (\nu n)(\overline{a}\langle n\rangle \,||\, \overline{c}\langle n\rangle)$, with $m \neq n$.

After the first transition we may both be receiving n or a name $m \neq n$:

$$(\{(n, 0[\overline{a}\langle n\rangle][\mathbf{0}])\}, \emptyset) \vdash P_1 \xrightarrow[1[\underline{b}(x).\overline{c}\langle n\rangle][\overline{c}\langle n\rangle]]{\underline{b}(n)} (\{(n, 0[\overline{a}\langle n\rangle][\mathbf{0}])\}, \{(n, 1)\}) \vdash (\mathbf{0} \,||\, \overline{c}\langle n\rangle).$$

$$(\{(n, 0[\overline{a}\langle n\rangle][\mathbf{0}])\}, \emptyset) \vdash P_1 \xrightarrow[1[\underline{b}(x).\overline{c}\langle n\rangle][\overline{c}\langle n\rangle]]{\underline{b}(m)} (\{(n, 0[\overline{a}\langle n\rangle][\mathbf{0}])\}, \{(m, 1)\}) \vdash (\mathbf{0} \,||\, \overline{c}\langle n\rangle).$$

In the first case, a subsequent output of n on channel c will not be an extrusion, since it happens after the input of n from the environment. In the second case it will, since this output is independent of the extrusion in transition 1. The input history $\underline{H}$ allows the (PAR_i) rules to distinguish between outputs that knew of n before the scope of n was opened and outputs that learned of n by receiving it after it was extruded.

Finally, if we again ignore histories and locations, the (COM_i) rules are the usual communication rules combined with the close rule, closing a scope previously opened by an (OPEN) rule. The combination is by abuse of notation, writing $(\nu\emptyset)P$ for P when there is the communication of a free name (in the same style to how the standard (CLOSE) rule for communication of a bound name). The location label is made into a pair, recording the two prefixes taking part in the communication. Looking at the histories, we discard any changes to histories formed in each component and only forward input histories from the sender to the receiver via the set $\underline{H}''$.

3.3 Correctness results

In this section we prove two correctness results for our semantics. The first correctness result shows that the standard interleaving, early operational semantics of pi-calculus (which we recall in Definition 3.3.2) can be obtained from the rules in Fig. 3.2 by ignoring the locations (Lemma 3.3.7) and histories (Lemma 3.3.4) and extracting only the scope extrusion from the histories. The result is stated as a bisimulation Corollary 3.3.10 from the two results of Proposition 3.3.8 which shows that every transition from pi-calculus is preserved in our semantics, and Proposition 3.3.9 which shows that our semantics does not introduce more transitions. The second correctness result from Theorem 3.3.11 shows that our semantics is a conservative extension of the one for CCS given in [MN92].

Definition 3.3.1 (generated transition system): The operational rules for pi-calculus that we gave in Figure 3.2 generate a transition system $TS_{psi} = (S, E, \mathcal{T})$ in the following way.

- The set of *states* contains pairs of a history and a process term, denoted $(\overline{H}, \underline{H}) \vdash P$:

$$S \triangleq \{(\overline{H}, \underline{H}) \vdash P \mid P \in \mathbf{Proc}, (\overline{H}, \underline{H}) \in \mathcal{H} \times \mathcal{H}\},$$

  with **Proc** as in Definition 3.2.1 and histories from Definition 3.2.11;

- $E = \mathbf{Ev}$ are the events from Definition 3.2.14 which label the transitions;

- $\mathcal{T} \subseteq S \times E \times S$ are the transitions from Definition 3.2.12.

For some particular process P we add an initial state $(\emptyset, \emptyset) \vdash P$ and work only with the transition system *reachable* from this initial state, denoted $TS_{psi}(P)$, and defined as the restriction of $TS_{psi}|_{S_P}$ to only the states

$$S_P = \{(\overline{H}, \underline{H}) \vdash P' \mid (\emptyset, \emptyset) \vdash P \to^* (\overline{H}, \underline{H}) \vdash P'\},$$

with $\to^*$ being the transitive closure of the transition relation $\mathcal{T}$.

Definition 3.3.2 (standard pi-calculus semantic rules): Denote by $\to_\pi$ the transitions obtained with the standard pi-calculus semantic [Qua99, MPW93] over the pi-calculus syntax from Definition 3.2.1. We recall these structural operational rules here, ignoring their (TAU) and (MATCH) rule.

$$\frac{n \notin fn((\nu x)P)}{\underline{a}(x).P \xrightarrow{\underline{a}(n)}_\pi P[x := n]}\ (\textsc{inp}) \qquad \frac{}{\overline{a}\langle n\rangle.P \xrightarrow{\overline{a}\langle\epsilon\rangle n}_\pi P}\ (\textsc{out})$$

$$\frac{P \,||\, !P \xrightarrow{\alpha}_\pi P'}{!P \xrightarrow{u}_\pi \alpha P'}\ (\textsc{bang}) \qquad \frac{\varphi_i.P_i \xrightarrow{\alpha}_\pi P_i'}{\Sigma\varphi_i.P_i \xrightarrow{\alpha}_\pi P_i'}\ (\textsc{sum}) \qquad \frac{P \xrightarrow{\alpha}_\pi P' \qquad bn(\alpha) \cap fn(Q) = \emptyset}{P \,||\, Q \xrightarrow{\alpha}_\pi P' \,||\, Q}\ (\textsc{par})$$

$$\frac{P \xrightarrow{(\nu n)\overline{a}\langle n\rangle}_\pi P' \qquad Q \xrightarrow{\underline{a}(n)}_\pi Q'}{P \,||\, Q \xrightarrow{\tau}_\pi (\nu n)(P' \,||\, Q')}\ (\textsc{close})$$

$$\frac{P \xrightarrow{\overline{a}\langle n\rangle n}_\pi P' \qquad Q \xrightarrow{\underline{a}(n)}_\pi Q'}{P \,||\, Q \xrightarrow{\tau}_\pi P' \,||\, Q'}\ (\textsc{com})$$

$$\frac{P \xrightarrow{\alpha}_\pi P' \qquad b \notin n(\alpha)}{(\nu b)P \xrightarrow{\alpha}_\pi (\nu b)P'}\ (\textsc{res}) \qquad \frac{P \xrightarrow{\overline{a}\langle\epsilon\rangle n}_\pi P' \qquad n \neq a}{(\nu n)P \xrightarrow{(\nu n)\overline{a}\langle n\rangle}_\pi P'}\ (\textsc{open})$$

Note our notation $(\nu n)\overline{a}\langle n\rangle$ instead of what normally in pi-calculus is denoted $\overline{a}(n)$ to signal that the name n is bound.

Definition 3.3.3 (standard pi generated transition system): For a pi-process P, the transition system reachable from P using the standard interleaving, early operational semantics from Definition 3.3.2 is denoted $TS_\pi = (S, P, \Lambda, \rightarrow_\pi)$ and defined similarly:

- The set of *states* contains just process terms as in Definition 3.2.1, i.e., $S \triangleq \{P \mid P \in \mathbf{Proc}\}$;
- The set of labels $\lambda \in \Lambda$ is defined by the grammar $\lambda ::= (\nu n)\overline{m}\langle n\rangle \mid \overline{m}\langle n\rangle \mid \underline{m}(n) \mid \tau$
- $\rightarrow_\pi \subseteq S \times \Lambda \times S$ are the transitions from Definition 3.3.2.

We prove that in our semantics the histories do not affect the enabling of a standard π transition. The statement of the Lemma 3.3.4 is on transitions directly derived from our semantic rules of Figure 3.2 but it extends to causal semantics transitions (i.e., split transitions) as in Definition 3.2.12 due to the Proposition 3.2.13.

Lemma 3.3.4 (histories in semantics): For a process P, if there exists a history $(\overline{H}, \underline{H})$ such that there exists a transition $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ then for any history $(\overline{H}_v, \underline{H}_v)$ we can find the same transition $(\overline{H}_v, \underline{H}_v) \vdash P \xrightarrow[u]{\alpha} (\overline{H}'_v, \underline{H}'_v) \vdash P'$.

Proof : We use induction on the length of the derivation tree for the given transition. We look at the last rule application (the root of the derivation tree), and show that we can apply it also to the different histories $(\overline{H}_v, \underline{H}_v)$.

The base cases for rules (IN) and (OUT) are easy since the histories are not influencing the decision of the transition (not the action label nor the location label). The (IN) rule updates the resulting input history though, but this has no bearing on our lemma.

We take now cases.

1. For rules (REP), (SUM), and (SCOPE), we see that a general history $H = (\overline{H}, \underline{H})$ is propagated, unchanged, from the conclusion of the rule to the hypotheses. This allows to apply the induction hypothesis thus allowing to replace the history with any history $(\overline{H}_v, \underline{H}_v)$. Moreover, the adjacent requirements of these rules (when any) do not involve the history information.

2. For rule (OPEN) the left-side history is just propagated to the hypothesis, and the adjacent requirements do not involve the history. This allows to apply the induction hypothesis as before. The output history is changed though.

3. For rules (PAR$_i$) the argument uses the induction hypothesis and also the fact that we can derive one transition with some history $(\overline{H}, \underline{H})$ from the process $P_0 \,||\, P_1$. We prove that we can derive the same transition from any other history and the same process $(\overline{H}_v, \underline{H}_v) \vdash P_0 \,||\, P_1$. From the fact that we can derive one transition it means we can derive from the component process $([\check{i}]\overline{H}, [\check{i}]\underline{H}) \vdash P_i \xrightarrow[u]{\alpha} (\overline{H}'_i, \underline{H}'_i) \vdash P'_i$. By the induction hypothesis this means we can derive the same transition from any output history, including the empty history as well as the history $[\check{i}]\overline{H}_v$. This means that for the outcome of the lemma we have the existence of the transition that is required by the (PAR$_i$). The other requirements of this rule do not influence its application, with one exception. In particular, the construction of $\overline{H}''$ is only used to update the output history on the right side of the derived transition.

   The exception is the fact that we need to check that no names from the other component process appear in the history changes (i.e., in $\tilde{b} = \mathsf{dom}(\overline{H}'''_i) \setminus \mathsf{dom}([\check{i}]\overline{H}_v)$). The intuition of $\tilde{b}$ is that it keeps the names that were opened, which in standard π are called bound names in the action label. However, these bound action names are derived only from the process, and the history does not influences them. In other words, the changes in the histories that $\tilde{b}$ records are the same for any history. Therefore, since we already know from the existing transition that $(\mathsf{dom}(\overline{H}'_i) \setminus \mathsf{dom}([\check{i}]\overline{H})) \cap fn(P_j) = \emptyset$ this would hold for our arbitrary history as well $\mathsf{dom}([\check{i}]\overline{H}_v)$. Formally this is proven using induction on the derivation as the following Corollary 3.3.5.

4. The same argument as for (PAR) works for (COM) rules, only that we need to apply two times the induction hypothesis.

□

Intuitively, the proof of Lemma 3.3.4 makes clear how only the process term is needed when deciding the existence of a transition, and the histories are only used to keep track of which names have been extruded, so that the (COM_i) rules can close the scope if needed. For the standard π calculus this is kept in the action label as the bound name of the action. Otherwise, histories are used in Definition 3.2.12 to split transitions.

Corollary 3.3.5: The change of the output histories in a transition does not contain names from any other components, and this change is the same irrespective of the left-side histories. Moreover, the (PAR_i) rules do not change the domains of the histories before and after the derived transition.

Proof: The base cases are for (IN) and (OUT) for which the output histories are not changed by the transition, thus making $\tilde{b} = \emptyset$, and thus proving the claim trivially.

For the induction cases (REP), (SUM), and (SCOPE), just apply the induction hypothesis trivially, since the histories are copied from the hypothesis of the rule.

The case for (OPEN) changes the output history by adding the name n. However, since this name was coming from under a restriction operator (ν) it means the name cannot appear in any other component of the process.

The case for (PAR_i) is more complex since this transition changes the output histories in two ways. First, names at the current location are replaced (i.e., $[i]\overline{H}$ are removed) with the ones that the component transition outputs (i.e., $i\overline{H}'_i$ are added). This respects the claim by applying the induction hypothesis. Second, it adds names in $i\overline{H}''$ which we want to prove that these do not appear outside $P_0 \,||\, P_1$. The set $\overline{H}''$ constructed in the rule contains a name $(n, u$ if this is output by the action α and (i) was output before by the other component P_j (i.e., $n \in \mathsf{dom}([j]\overline{H})$), and (ii) the same name was not output before by a sub-process under the current location nor received by a sub-process at the current location. Part (ii) is fine, however, for part (i) the name n must come from under a restriction operator (ν) in P_1; but this means that it does not exist outside this, and thus in any other component than the current parallel process.

The case for (COM_i) follows similar arguments. □

Corollary 3.3.6: For any of the rules, if $\mathsf{dom}(\overline{H} \cup \underline{H}) \cap bn(P) = \emptyset$ holds before the rule (ensured by the requirement in the caption of Figure 3.2) then the same holds after the rule.

Proof: The proof is done by induction on the derivation tree.

The base case for (OUT) rule is trivial since this does not change the histories. Whereas for the (IN) rule the input history is changed by adding the name that is received. But this is fine since this name cannot appear bound in neither the process before nor the one after.

The induction case for the rules (SCOPE), (REP), and (SUM) is immediate by the induction hypothesis since the histories below the derivation line are the same as the ones above.

The (OPEN) rule adds to the output histories the name n but removes the restriction operator on this name from the process before the transition. The induction hypothesis then finishes the proof.

For the (COM) rules the argument is split in two parts: (1) for the set $\tilde{b}$ and (2) for the set $j\underline{H}''$, knowing that $\mathsf{dom}(\overline{H} \cup \underline{H}) \cap bn(P_0 \,||\, P_1) = \emptyset$. To prove the statement for the process on the right of the transition we observe that this has extra bound names the $\tilde{b}$.

1. We prove that names from $\tilde{b}$ do not appear in $\mathsf{dom}(\overline{H} \cup \underline{H})$ and neither in $\underline{H}''$. Since $\tilde{b} = \mathsf{dom}(\overline{H}'_i) \setminus \mathsf{dom}([\check{i}]\overline{H})$ it means that $\tilde{b}$ is fresh for the histories $\overline{H} \cup \underline{H}$ and is not brought along in the histories on the right side of the derived transition. Moreover, $\tilde{b}$ does not appear in the rest of the parallel components since the rule requires $\tilde{b} \cap n(P_j) = \emptyset$.

2. Second, we prove that names from $j\underline{H}''$ do not appear in neither $\tilde{b}$ nor in $bn(P)$, thus not in the bound names of the process from the right side of the derived transition. It is easy to see that $\underline{H}''$ contains only names that appear in $\overline{H} \cup \underline{H}$, which by induction hypothesis they are not in $bn(P)$. Moreover, since $\tilde{b}$ are not in $\mathsf{dom}(\overline{H} \cup \underline{H})$ then are not in $\underline{H}''$ either.

Take one rule ($\textsc{par}_0$), as the argument is analogous for the other, and we prove that $\mathsf{dom}(((\overline{H} \setminus [0]\overline{H}) \cup 0\overline{H}'_0 \cup 0\overline{H}'' \cup (\underline{H} \setminus [0]\underline{H}) \cup 0\underline{H}'_0)) \cap bn(P'_0 \,||\, P_1) = \emptyset$. We take each set from the sequence of unions. For $\overline{H} \setminus [0]\overline{H}$ since it removes all names starting with 0 this will not contain names from P'_0 either. From the assumption on the left of the transition, we know that the remaining names are not in P_1 either. The same argument goes for $\underline{H} \setminus [0]\underline{H}$. Now use the induction hypothesis to deduce that $\overline{H}'_0 \cap bn(P'_0) = \emptyset$ as well as $\underline{H}'_0 \cap bn(P'_0) = \emptyset$. Note that adding the digit 0 keeps the domains the same. We are left with $\overline{H}''$ which we know it is a subset of $[1]\underline{H}$ and does not already appear to have been part of the histories coming from the P_0 component. Therefore, by the assumption we also have that $\overline{H}''$ does not have names in the bound names of neither P_1 nor in P'_0. □

In order to compare our semantics with the standard one for π calculus we still need to show how the location labels can be ignored, since these can be determined solely from the two process terms involved in the transition.

Lemma 3.3.7 (location labels are determined): Whenever we have a transition $(\overline{H}, \underline{H}), P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}'), P'$ then the location label u is unique, i.e., determined by P and P'.

Proof: We generate the location u in the derivation tree of the transition. We assume inductively that the shorter tree generates a unique label. We then look at the structure of P and P', and consider cases on the last rule applied at the root of the derivation tree.

- If $P = \overline{a}\langle n\rangle.P'_0$ or $P = \underline{a}(n).P'_0$ then we know from ($\textsc{out}$) respectively ($\textsc{in}$) that $u = [\overline{a}\langle n\rangle.P'_0][P'_0]$ respectively $u = [\underline{a}(n).P'_0][P'_0]$ are unique. This forms the base case of the induction.
- If $P = P_0 \,||\, P_1$ and $P' = P'_0 \,||\, P_1$ then we know that ($\textsc{par}_0$) rule is applied at the root and it adds to the location label $u = 0v$ where v is the unique label obtained from the proof of the P_0 branch. Similarly we find $u = 1v$ if $P' = P_0 \,||\, P'_1$ using ($\textsc{par}_1$).
- If $P = P_0 \,||\, P_1$ and $P' = P'_0 \,||\, P'_1$ we know that ($\textsc{com}$) rule is applied and that $u = \langle 0u_0, 1u_1\rangle$, which is uniquely obtained by continuing up each branch finding the unique u_0 for the P_0 part and u_1 for the P_1 part.
- If P is of any other process types then the respective application of the remaining rules just copy the location labels.

□

Proposition 3.3.8 (Pi-calculus semantics is preserved): For a pi-process P whenever we have a transition $P \xrightarrow{\alpha}_\pi P'$ then we have in our semantics a transition $(\emptyset, \emptyset) \vdash P \xrightarrow[u]{\alpha'} (\overline{H}, \underline{H}) \vdash P'$ for some determined location label u and history $(\overline{H}, \underline{H})$, and when $\alpha = (\nu n)\overline{m}\langle n\rangle$ then $\alpha' = \overline{m}\langle n\rangle$ otherwise $\alpha' = \alpha$.

Note that Proposition 3.3.8 claims the transition starting with the empty history, but then Lemma 3.3.4 allows any histories to be used. The Lemma 3.3.7 tells how the location label u is determined.

Proof: We use induction on the derivation tree and show how for each rule application in the pi-calculus we find rules applied in our semantics which return the required process.

It is easy to see how when the pi-rules ($\textsc{inp}$) and ($\textsc{out}$) are applicable, then the respective rules in Figure 3.2 are also applicable to the empty histories, and the output process as well as the action label are the same as in the pi-rule. In the rules from Figure 3.2 we update the input history in the ($\textsc{in}$) rule, and create the respective location labels.

The pi-rule ($\textsc{bang}$) is the same as our rule ($\textsc{rep}$). The same for the pi-rule ($\textsc{res}$) with its requirements which appear the same in our rule ($\textsc{scope}$). The pi-rule ($\textsc{sum}$) is as our rule. To all these the resulting processes are exactly the same. Moreover, the histories are not even changed, neither the location labels.

The pi-rules ($\textsc{par}$), ($\textsc{com}$), and ($\textsc{open}$) are manipulating the same processes as our respective rules from Figure 3.2, and the respective requirements can be correlated. In our case we manipulate histories and construct location labels.

In particular, the (OPEN) rule from Figure 3.2 has the same requirement on the name $n \neq a$ and it changes the output history to record the bound name that was output from under a scope, thus opening it. In the pi-rule this bound name is recorded in the action label, which in their case changes to $(\nu n)\overline{a}\langle n\rangle$. This is why the statement of the proposition captures this distinction between action labels.

For the pi-rules (PAR) the requirement $bn(\alpha) \cap fn(Q) = \emptyset$ is captured in our (PAR$_i$) rules by $\tilde{b} \cap fn(P_j) = \emptyset$ where $\tilde{b}$ is stored in the output history.

For the pi-rule (COM) note that the output action has no bound name, and thus no restriction operator is surrounding the parallel process. In our (COM) rules this implies that $\tilde{b} = \emptyset$, and we use the syntactic sugar $(\nu\emptyset)P_0 \,||\, P_1$ instead.

Otherwise, for the pi-rule (CLOSE) in our rules the $\tilde{b} \neq \emptyset$ and thus the restriction operator will be exactly as in the pi-rule.

One last aspect that we need to show is that the histories do not prohibit transitions which otherwise are allowed in the pi-calculus transition system. This was done in Lemma 3.3.4. □

Proposition 3.3.9 (no extra transitions): For a process P whenever we have a transition in our semantics $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ then we find the transition in the standard pi-calculus semantics $P \xrightarrow{\alpha'}_{\pi} P'$ with $\alpha' = (\nu n)\alpha$ if $\mathsf{dom}(\overline{H}')\backslash\mathsf{dom}(\overline{H}) = \{n\}$ and $\alpha' = \alpha$ otherwise.

Proof: Consider our operational rules from Figure 3.2 and use induction on the derivation tree to show that for each rule applied in our semantics when we remove the histories and location labels than either the same rule is applicable in the pi-calculus or we find other rules applied which return the required process. The reasoning is very similar to what we did in the proof of Proposition 3.3.8.

The only special aspect is the fact that we need to extract from the output history the action label when this needs to have a bound name, as coming from the (OPEN) rule. □

Corollary 3.3.10 (conformance wrt. pi-calculus): For some process P the generated transition system reachable from this process in the standard pi-semantics $TS_{\pi}(P)$ and our semantics $TS_{psi}(P)$ are bisimilar.

We end by noting that the non-interleaving semantics is a conservative extension of the one for CCS given in [MN92]. To this end, consider as equivalent to CCS the sub calculus of the pi-calculus obtained by allowing only input and output prefixes in which the subject and object are the same, i.e. of the form $\underline{n}(n)$ and $\overline{n}\langle m\rangle n$, referred to as the CCS subset. In this case it is easy to see that the output histories and link dependencies D are always empty and thus the independence relation and events coincide with the structural independence and events.

Theorem 3.3.11: For the CCS subset of the pi-calculus, the non-interleaving semantics of Figure 3.2 is bisimilar to the non-interleaving semantics for CCS given in [MN92].

Proof: To be precise, the CCS calculus from [MN92] builds processes using the grammar:

$$P^{CCS} ::= \Sigma_{i\in I}\varphi_i.P_i^{CCS} \;|\; (\nu n)P^{CCS} \;|\; P^{CCS} \,||\, Q^{CCS} \;|\; x \;|\; rec\, x.P^{CCS} \qquad \text{with } \varphi \in \{\overline{\alpha}, \underline{\alpha} \mid \alpha \in \mathcal{N}\}.$$

They use recursive definitions instead of our replication construct; but these are encodable, e.g., see [AGPV07, Def.6], so we will not be concerned with this. Otherwise, the difference is in their action prefixes which are only names of the two kinds output/input. Therefore, this forms a subset of the pi-calculus that we considered in Definition 3.2.1 when we allow only output/input with the same subject and object, i.e., $\overline{n}\langle n\rangle \setminus \underline{n}(n)$. See a nice comparisons of CCS and pi-calculus in this sense as Psi-calculi instances in [BJPV11]. The proof is then formed of the following observations.

1. the output histories;
2. the link dependencies D are always empty, thus events are never split;

3. the independence relation and events coincide with the structural independence and events;
4. any transition in the CCS semantics is matched by the same transition in our semantics of Figure 3.2, and the other way around.

To prove these we investigate our semantic rules, and discuss how these relate to the semantic rules from [MN92], which we do not reproduce here as they are revealed from our comparative discussions.

1. For the first statement we look at the transition rules (PAR_i) and (OPEN), as these are the only rules that add to the output histories. The rule (IN) adds to the input history, which is used in the (PAR_i) rules to determine what to add to the output histories. The (OPEN) rule is not applicable since it requires for an output prefix $\overline{a}\langle n\rangle$ that $n \neq a$, which does not hold for the prefixes that are allowed in the CCS instance. The extra names added in the (PAR_i) rules depend on previously added names by the (OPEN) rule, i.e., see the condition for forming the $\overline{H}''$ set in (PAR_i) rules. Since these are empty, then (PAR_i) rules do not add any new names either.
2. Since histories are empty, and in Definition 3.2.12 the link dependencies set D is created from the histories as $D \subseteq \overline{H}$, this statement is trivially proven.
3. This is easy to see from the previous part 2 since Ds are always empty, then in Definition 3.2.14 an event $\big((\alpha, u), D\big) = (\alpha, u)$ as in Definition 3.2.6. Also, the split transition from which the causal events are extracted $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u,\emptyset]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ are the same as the transitions $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ given by our semantic rules of Figure 3.2. As the link dependencies are empty, then the second part of the definition of independence relation in Definition 3.2.14 is trivial, thus I becoming just the structural independence I_{CCS} as in Definition 3.2.7, which is that of CCS from [MN92].
4. We already observed that (OPEN) is never applicable, which implies that the "close" part of the (COM_i) rules is not applicable (i.e., when the $\tilde{b} \neq \emptyset$ and thus a restriction operation is moved from the parallel component with the output prefix to the whole parallel composition process). Otherwise, the remaining (COM_i) rules are the same as in [MN92], building the same location label, since our restriction about names is vacuously satisfied for CCS.

   Also note that the (IN) and (OUT) rules are always applicable as the immediate rule on top of the (SUM) rule. Therefore, we can view these three rules as the single SUM rule from [MN92] which is also performing the action, and exhibiting the respective output/input action name on the transition. However, the location label that we build in this case contains only the branch process, whereas in [MN92] it contains the whole summation process. This is fine for our proofs, and does not interfere with the deduced transitions.

   The (SCOPE) rule is the same as the RES rule from [MN92], also with the required restriction, only that in the location label we do not need to record the name that is bound by the restriction operator as [MN92] needs.

   For the (PAR_i) rules our restriction on names is vacuously satisfied, and the location label is updated exactly as in [MN92]. We do not discuss the (REP) rule.

   The above observations make it clear that whenever from a P_{CCS} process one can derive a transition in the semantics of [MN92], we can also derive the same transition, with the same action label, and analogous location label in our semantics. The location label in our case does not contain bound names, and contains in the last process part of the label only the summation branch. However, these location labels do not contribute in [MN92] to deciding which transitions can be derived. The opposite is also true, i.e., in our semantics we cannot derive a transition which the semantics of [MN92] cannot derive also.

□

3.4 Labelled asynchronous transition systems from early operational semantics of pi-calculus

In this section we show that the operational semantics, events and independence relation given for the pi-calculus in the previous section yields a labelled asynchronous[1] transition system (LATS) as defined in [Bed88, Shi85b, WN95, Hil99] and recalled in Definition 3.4.1. LATS are known to satisfy the stability property, that is, every event depends on a unique set of events, and unfold to standard labelled prime event structures [WN95, Ch.7].

Definition 3.4.1: A *labelled asynchronous transition system* is a tuple $LATS = (S, i, E, \mathcal{T}, I, lab, \mathcal{A})$ where

- $(S, i, E, \mathcal{T})$ is a transition system with S the set of *states* and i an *initial state*, E a set of *events*, and $\mathcal{T} \subseteq S \times E \times S$ the *transition relation*;
- $lab : E \to \mathcal{A}$ is a *labelling* map from the set of events to the *action set* $\mathcal{A}$;
- $I \subseteq E \times E$ is an irreflexive, symmetric independence relation, satisfying:
  1. $e \in E \;\Rightarrow\; \exists s, s' \in S : (s, e, s') \in \mathcal{T}$;
  2. $(s, e, s') \in \mathcal{T} \wedge (s, e, s'') \in \mathcal{T} \;\Rightarrow\; s' = s''$;
  3. $e_1 I e_2 \wedge \{(s, e_1, s_1), (s, e_2, s_2)\} \subseteq \mathcal{T} \;\Rightarrow\; \exists s_3 : \{(s_1, e_2, s_3), (s_2, e_1, s_3)\} \subseteq \mathcal{T}$;
  4. $e_1 I e_2 \wedge \{(s, e_1, s_1), (s_1, e_2, s_3)\} \subseteq \mathcal{T} \;\Rightarrow\; \exists s_2 : \{(s, e_2, s_2), (s_2, e_1, s_3)\} \subseteq \mathcal{T}$.

The last two conditions ensure that independent events always form interleaving diamonds and imply *stability*, i.e. unique cause of events.

Theorem 3.4.2 (LATS for pi): The semantic rules for the pi-calculus that we gave in Fig. 3.2 generate a labelled asynchronous transition system $LATS(P) = (\mathbf{Proc}, (\emptyset, \emptyset) \vdash P, \mathbf{Ev}, \mathcal{T}, I, lab, \mathcal{A})$ for a pi-process P where

- $(\mathbf{Proc}, (\emptyset, \emptyset) \vdash P, \mathbf{Ev})$ is the generated transition system $TS_{psi}(P)$ reachable from P as in Definition 3.3.1 with **Proc** from Definition 3.2.1, histories from Definition 3.2.11, events from Definition 3.2.14, and transitions from Definition 3.2.12;
- I is the relation from Definition 3.2.14,
- $lab\big((\alpha, u), D\big) = \alpha$,
- $\alpha \in \mathcal{A}$ is the set of labels generated by the grammar $\alpha ::= \overline{a}\langle n \rangle \mid \underline{a}(n) \mid \tau$

To prove this we need several intermediary results and definitions. The following lemma states that the transition system is event deterministic, i.e. that it satisfies property 2. of Def. 3.4.1.

Lemma 3.4.3 (event determinism): For any two transitions $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ and $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\overline{H}'', \underline{H}'') \vdash P''$ then $(\overline{H}', \underline{H}') = (\overline{H}'', \underline{H}'')$ and $P' = P''$.

Proof: The events that we are interested in are the ones in **Ev** obtained through splitting in Definition 3.2.14. But each split event is obtained from a transition derived with the rules from Figure 3.2 and all splits make transitions between the same pairs of processes and histories. Therefore we are not interested in the splits but in the kernel event, which is formed of the first two elements of the three-tuple event, i.e., the action name and the location where the event appears. In consequence, we show that for some arbitrary event and process (together with some history), there is a unique operational rule that is applicable. This results in a unique transitions, as in the statement of the lemma.

The histories are not contributing to the decision whether an operational rule is applicable or not; and neither do the histories influence how the resulting process looks like. These facts have been established in Proposition 3.3.4.

For the rest we use induction on the structure of the location label and the process P, thus identifying which of the operational rules applies for the particular reduction. The induction is double, depending on how the location label looks like, cf. Proposition 3.2.4, i.e., either of type $(\alpha, s[P_1][P_1'])$ or $(\tau, s\langle s_0[P_0][P_0'], s_1[P_1][P_1']\rangle)$. We have two base cases.

[1] asynchronous here refers to non-interleaving, not the style of communication.

1. When $e = (\alpha, [P_1][P_1'])$ only rules (IN) or (OUT) are applicable. Depending on the structure of the process P, or for the same purpose, depending on α, only one of the two rules applies. The outcome is then determined, thus having a unique resulting process. The histories are also determined, i.e., for the (OUT) rule they are the same as on the left of the transition, whereas for the (IN) rule the input history is changed with the name from the action α.

2. When $e = (\tau, \langle 0[P_0][P_0'], 1[P_1][P_1']\rangle)$ (i.e., when the location label is a minimal communication location) only the (COM) rule applies, and the outcome process is unique with respect to the original process P. If $P = (\nu b)Q$ then the rule (OPEN) also applies and changes the output action to one having a bounded name b, and the resulting process is also restricted as $(\nu b)Q'$ with Q' determined by the communicating processes. Otherwise, if P is only a parallel composition, then the resulting process is not under a restriction.

For the induction case we consider that the event has the location

$$ss'[P_0][P_0'] \quad \text{or} \quad ss'\langle 0s_0[P_0][P_0'], 1s_1[P_1][P_1']\rangle$$

with $s \in \{0,1\}$, and any of the s', s_0, s_1 may be empty. We take cases after s and for each case we look at the structure of P and α to determine the rule used. It turns out that each time only one rule applies. Moreover, each rule changes the histories from the left of the transition in a determined way, ensuring the history equality from the statement of the lemma.

1. For the case when $s = 0$ we have that

   (a) if $P = P_0 \,||\, P_1$ then the only rule that could have been applied to generate this transition is (PAR$_0$). From the induction hypothesis we know that the required transition for this rule, which applies to a shorter location label, produces the unique process P_0', which when composed in parallel produces a unique outcome process $P_0' \,||\, P_1$.

   (b) if $P = !P_1$ then only rule (REP) applies to produce $Q \,||\, !Q$. Further up the derivation tree one of the (PAR) rules apply. When $s = 0$ the (PAR$_0$) rule produces the unique $P' \,||\, !Q$, whereas for $s = 1$ then $!Q$ reduces uniquely (by the induction hypothesis) to P'.

   (c) if $P = (\nu n)P_1$ then

      i. if $\alpha = \overline{a}\langle n\rangle n$ then only rule (OPEN) applies,

      ii. otherwise only (SCOPE) is applicable.

   Further up in the derivation tree we can apply the induction hypothesis on the same location label, but a smaller process (without the restriction operator), to produce a unique process, thus the whole derivation is unique.

2. For the case when $s = 1$ we have that

   - if $P = P_0 \,||\, P_1$ then only rule (PAR$_1$) applies, and an argument analogous to 1a goes through.
   - if $P = !P_1$ or $P = (\nu n)P_1$ then use analogous arguments as for 1b respectively 1c.

3. For all location labels and action labels if $P = \Sigma_{i\in I} : \varphi_i.P_i$ then the unique rule application is (SUM). Even if two branches of the sum induce a transition with the same action name but different output process, these will be considered as different events because of the location label containing different processes at the end. Therefore, the nondeterminism no longer exists here. The induction hypothesis is applicable further up on the derivation tree to the smaller label as well as smaller process, to produce a unique outcome.

For the location label $ss'\langle 0s_0[P_0][P_0'], 1s_1[P_1][P_1']\rangle$ an inductive argument as before reduces the s and s' to the case when they are empty. In this case the only rule that could have created this label is the (COM) rule. This implies that $P = Q_0 \,||\, Q_1$. Using the previous case on Q_0 and location $s_0[P_0][P_0']$ we know that Q_0 uniquely reduces to Q_0', and similarly for Q_1 and the location $s_1[P_1][P_1']$. Thus $Q_0 \,||\, Q_1$ reduces uniquely to $Q_0' \,||\, Q_1' = P'$. $\square$

To prove that the transition system from Theorem 3.4.2 satisfies the last two (diamond) properties of a labelled asynchronous transition system we follow the approach from [MN92].

The following partial function makes precise how a sequence $s \in \{0,1\}^*$ identifies a sub-process, called the *component*, in a process.

Definition 3.4.4 (components): Define inductively the partial function

$$Comp : \{0,1\}^* \times \mathbf{Proc} \rightharpoonup \mathbf{Proc}$$

1. $Comp(\epsilon, P) = P,$ when $P \neq\, !P_1$ and $P \neq (\nu n)P_1$ (and ϵ is the empty string)
2. $Comp(0s, P_0 \,||\, P_1) = Comp(s, P_0)$
3. $Comp(1s, P_0 \,||\, P_1) = Comp(s, P_1)$
4. $Comp(s, (\nu n)P) = Comp(s, P)$
5. $Comp(s, !P) = Comp(s, P \,||\, !P)$

Corollary 3.4.5: For any $s, s' \in \{0,1\}^*$ and any process P, whenever $Comp$ is defined, we have

$$Comp(s, Comp(s', P)) = Comp(s's, P).$$

Proof: The proof follows from Definition 3.4.4 by induction on the structure of s'. The base case for $s' = \varepsilon$ is easy by using Definition 3.4.4(1); and when $P =!P_1$ or $P = (\nu n)P_1$ then just apply the respective definitions.

Take $s' = 0s'_0$ for which $Comp$ is defined when P is one of $P_0 \,||\, P_1$, $(\nu n)P_0$, or $!P_0$. For $P = P_0 \,||\, P_1$ from Definition 3.4.4(2) we have that $Comp(0s'_0, P_0 \,||\, P_1) = Comp(s'_0, P_0)$, which implies that the left part of the lemma equality becomes $Comp(s, Comp(s'_0, P_0))$. The right part becomes $Comp(s'_0 s, P_0)$ by applying Definition 3.4.4(2) as $Comp(0s'_0 s, P_0 \,||\, P_1) = Comp(s'_0 s, P_0)$. The equality of these last two formulas is given by the induction hypothesis. In the case when $P =!P_0$ then we first apply Definition 3.4.4(5) and then we follow the above reasoning verbatim. For the case for $P = (\nu n)P_0$ we first use Definition 3.4.4(4) then follow as above.

The case for $s' = 1s'_1$ is analogous, using Definition 3.4.4(3) for $P = P_0 \,||\, P_1$ and Definition 3.4.4(4) for $P = (\nu n)P$.

Note that no matter what the structure of s is, the $Comp$ is defined when $P =!P_0$. This then reduces to a process where only the cases Definition 3.4.4(2) and (3) could further be applicable. □

From any transition we can deduce the transition in the immediate component responsible for the derivation.

Lemma 3.4.6 (decomposing transitions): For $s \in \{\varepsilon, 0, 1\}$ and $s' \in \{\varepsilon, 0, 1\}^*$ we have

$$(\overline{H}, \underline{H}) \vdash P \xrightarrow[ss'u_\varepsilon]{\alpha} (\overline{H}', \underline{H}') \vdash P' \quad \Rightarrow \quad ([\check{s}]\overline{H}, [\check{s}]\underline{H}) \vdash Comp(s, P) \xrightarrow[s'u_\varepsilon]{\alpha'} (\overline{H}'', \underline{H}'') \vdash Comp(s, P'),$$

with u_ε either $[P_u][P'_u]$ or $\langle 0s_0s'_0[P_0][P'_0], 1s_1s'_1[P_1][P'_1]\rangle$, and depending on the case we have the following extra properties:

1. when $s = 0$ we have $\alpha' = \alpha$ and $(\mathsf{dom}(\overline{H}'') \backslash \mathsf{dom}([\check{s}]\overline{H})) \cap fn(Comp(1, P)) = \emptyset$;
2. when $s = 1$ we have $\alpha' = \alpha$ and $(\mathsf{dom}(\overline{H}'') \backslash \mathsf{dom}([\check{s}]\overline{H})) \cap fn(Comp(0, P)) = \emptyset$;
3. when $s = \varepsilon$ and $P = (\nu\tilde{n})P_1$ and $P_1 \neq (\nu\tilde{m})P_2$, for $\tilde{n}$ and $\tilde{m}$ non-empty,[2]
   we either have ($\tilde{n} \cap n(\alpha) = \emptyset$ and $\alpha' = \alpha$) or ($\exists b \in \tilde{n} : \alpha = \overline{a}\langle b \rangle b$ and $\alpha' = \overline{a}\langle \epsilon \rangle b$ with $a \notin \tilde{n}$);
4. when $s = \varepsilon$ and $P =!P_1$ we have $\alpha' = \alpha$.

[2] By applying alpha conversion we can assume that all restrictions are appearing at the front of the process term, and thus we denote by $(\nu\tilde{n})$ the list of all bound names of P.

Proof: We take cases after $s \in \{\epsilon, 0, 1\}$.

- When $s = 0$ the rule that adds 0 to the location label is (PAR_0), therefore we work with the transition $(\overline{H}, \underline{H}) \vdash P_0 \,||\, P_1 \xrightarrow[ss'u_\epsilon]{\alpha} (\overline{H}', \underline{H}') \vdash P_0' \,||\, P_1$. The rule requires that transition $([\check{0}]\overline{H}, [\check{0}]\underline{H}) \vdash P_0 \xrightarrow[s'u_\epsilon]{\alpha} (\overline{H}'', \underline{H}'') \vdash P_0'$ exists and that $\tilde{b} \cap fn(P_1) = \emptyset$, with $\tilde{b} = \mathsf{dom}(\overline{H}'') \backslash \mathsf{dom}([\check{0}]\overline{H})$. Applying Definition 3.4.4 for deriving components, the above transform into the expected result, i.e.: $([\check{0}]\overline{H}, [\check{0}]\underline{H}) \vdash Comp(0, P_0 \,||\, P_1) \xrightarrow[s'u_\epsilon]{\alpha'} (\overline{H}'', \underline{H}'') \vdash Comp(0, P_0' \,||\, P_1)$ with $\alpha' = \alpha$ and $(\mathsf{dom}(\overline{H}'') \backslash \mathsf{dom}([\check{0}]\overline{H})) \cap fn(Comp(1, P)) = \emptyset$.
  If $s = 0$ and $P = !P_s$ the application of the (REP) rule implies that we have a transition from $(\overline{H}, \underline{H}) \vdash P_s \,||\, !P_s \xrightarrow[ss'u_\varepsilon]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ with the same action, history, and label as the given transition of the statement of the lemma. On this we can now use the (PAR_0) rule and apply the same argument as above to deduce the transition $([\check{0}]\overline{H}, [\check{0}]\underline{H}) \vdash Comp(s, P_s \,||\, !P_s) \xrightarrow[s'u_\varepsilon]{\alpha} (\overline{H}'', \underline{H}'') \vdash Comp(s, P')$. By Definition 3.4.4(5) we know that $Comp(s, P) = Comp(s, P_s \,||\, !P_s)$, which ends this case.
- When $s = 1$ we follow the same argument as above using (PAR_1) instead of (PAR_0).
- When $s = \epsilon$ we consider the three rules (OPEN), (SCOPE), and (REP), where we do not add anything to the location label. The structure of P and of α determine which rule is applicable. We omit the histories when obvious from the context so to not clutter the text.
  - Consider the application of (OPEN) to process $P = (\nu n)P_0$ and action $\alpha = \overline{a}\langle n\rangle n$ where $a \neq n$. We thus work with a transition $(\nu n)P_0 \xrightarrow[\epsilon s'u_\epsilon]{\alpha} P_0'$ for which the rule ensures the existence of the transition $P_0 \xrightarrow[s'u_\epsilon]{\overline{a}\langle\epsilon\rangle n} P_0'$. As $Comp(\epsilon, (\nu n)P_0) = P_0$ and $Comp(\epsilon, P_0') = P_0'$ we get the expected transition $Comp(\epsilon, (\nu n)P_0) \xrightarrow[s'u_\epsilon]{\overline{a}\langle\epsilon\rangle n} Comp(\epsilon, P_0')$ with the actions as expected $\alpha = \overline{a}\langle n\rangle n$ and $\alpha' = \overline{a}\langle\epsilon\rangle n$.
  - Consider the application of (SCOPE) to a process $P = (\nu n)P_0$, thus working with a transition $(\nu n)P_0 \xrightarrow[\epsilon s'u_\epsilon]{\alpha} (\nu n)P_0'$. The rule ensures that $n \in n(\alpha)$ and that we have a transition $P_0 \xrightarrow[s'u_\epsilon]{\alpha} P_0'$. Applying Definition 3.4.4 we get the expected transition $Comp(\epsilon, (\nu n)P_0) \xrightarrow[s'u_\epsilon]{\alpha} Comp(\epsilon, (\nu n)P_0')$, with the action $\alpha' = \alpha$.
  - Consider the application of (REP) to a process $P = !P_0$, thus working with a transition $!P_0 \xrightarrow[\epsilon s'u_\epsilon]{\alpha} P'$. The rule ensures the existence of the transition $P_0 \,||\, !P_0 \xrightarrow[s'u_\epsilon]{\alpha} P'$. As $Comp(\epsilon, !P_0) = Comp(\epsilon, P_0 \,||\, !P_0) = P_0 \,||\, !P_0$ and $Comp(\epsilon, P') = P'$, we get the expected transition $Comp(\epsilon, !P_0) \xrightarrow[s'u_\epsilon]{\alpha} Comp(\epsilon, P')$.

□

Corollary 3.4.7: Applying several times Lemma 3.4.6, we can extend s to be a string of location components: $s \in \{0, 1\}^*$.

From any communication transition we can then recover the transitions in the components identified by the location labels.

Lemma 3.4.8 (decomposing communications): For location strings $s, s_0, s_1, s_0', s_1' \in \{0, 1\}^*$ the following holds

$$\text{if} \quad (\overline{H}, \underline{H}) \vdash P \xrightarrow[s\langle 0s_0s_0'[P_0][P_0'], 1s_1s_1'[P_1][P_1']\rangle]{\tau} (\overline{H}', \underline{H}') \vdash P' \quad \text{then}$$

$$([\check{s}_l]\overline{H}, [\check{s}_l]\underline{H}) \vdash Comp(s0s_0, P) \xrightarrow[s_0'[P_0][P_0']]{\alpha} (\overline{H}'', \underline{H}'') \vdash Comp(s0s_0, P') \quad \text{and}$$

$$([\check{s_r}]\overline{H}, [\check{s_r}]\underline{H}) \vdash Comp(s1s_1, P) \xrightarrow[s_1'[P_1][P_1']]{\overline{\alpha}} (\overline{H}''', \underline{H}''') \vdash Comp(s1s_1, P')$$

where $s_l = s0s_0$, $s_r = s1s_1$, and $\overline{\alpha}$ notation is defined as $\overline{\overline{a}\langle\tilde{n}\rangle n} = \underline{a}(n)$ and $\overline{\underline{a}(n)} = \overline{a}\langle\tilde{n}\rangle n$.

Proof: We first make use of Lemma 3.4.6 to simplify to only considering $s = \varepsilon$. Then we work with the labels $0s_0$ and $1s_1$ and make two arguments in parallel.

Therefore, we consider P of the form $P_0 \,||\, P_1$ and apply the (COM) rule at the root of the transition derivation tree. This implies the existence of the two transitions $([\check{0}]\overline{H}, [\check{0}]\underline{H}) \vdash P_0 \xrightarrow[s_0s_0'[P_0][P_1']]{\alpha} (\overline{H}'_0, \underline{H}'_0) \vdash P_0'$ and $([\check{1}]\overline{H}, [\check{1}]\underline{H}) \vdash P_1 \xrightarrow[s_1s_1'[P_1][P_1']]{\overline{\alpha}} (\overline{H}'_1, \underline{H}'_1) \vdash P_1'$, with $Comp(0, P) = P_0$ and $Comp(1, P) = P_1$. Moreover, α and $\overline{\alpha}$ must be of the forms $\overline{a}\langle\tilde{n}\rangle m$ and $\underline{a}(m)$, and $P' = (\nu\tilde{b})(P_0' \,||\, P_1')$, with $\tilde{b} = \mathsf{dom}(\overline{H}'_0)\backslash\mathsf{dom}([\check{0}]\overline{H})$. We know that the channel name a must be the same in both α's and the output name m is the same as the received name on the input side. Moreover, if $\tilde{b} \neq \epsilon$, the only time a name can be added to an output action label is by the (OPEN) rule, which opens the scope of b and thus we must have $m = b$.

Apply now Lemma 3.4.6 (in fact Corollary 3.4.7 when the labels are strings) to the two transitions to obtain

$$([\check{s_0}][\check{0}]\overline{H}, [\check{s_0}][\check{0}]\underline{H}) \vdash Comp(s_0, P_0) \xrightarrow[s_0'[P_0][P_1']]{\alpha} (\overline{H}'', \underline{H}'') \vdash Comp(s_0, P_0')$$

and

$$([\check{s_1}][\check{1}]\overline{H}, [\check{s_1}][\check{1}]\underline{H}) \vdash Comp(s_1, P_1) \xrightarrow[s_1'[P_1][P_1']]{\overline{\alpha}} (\overline{H}''', \underline{H}''') \vdash Comp(s_1, P_1'),$$

which by Corollary 3.4.5 are the same as $([0\check{s}_0]\overline{H}, [0\check{s}_0]\underline{H}) \vdash Comp(0s_0, P) \xrightarrow[s_0'[P_0][P_1']]{\alpha} (\overline{H}'', \underline{H}'') \vdash Comp(s_0, P_0')$ respectively $([1\check{s}_1]\overline{H}, [1\check{s}_1]\underline{H}) \vdash Comp(1s_1, P) \xrightarrow[s_1'[P_1][P_1']]{\overline{\alpha}} (\overline{H}''', \underline{H}''') \vdash Comp(s_1, P_1')$.

To see how the right components become the ones from the statement of the lemma apply Definition 3.4.4(4) to P' with the respective location labels $0s_0$ and $1s_1$, having e.g.,

$$Comp(0s_0, P') = Comp(0s_0, (\nu\tilde{n})(P_0' \,||\, P_1')) = Comp(0s_0, P_0' \,||\, P_1') = Comp(s_0, P_0').$$

□

Conversely, we can lift a transition from a component to the whole process, when it does not enter a communication.

Lemma 3.4.9 (composing transitions): For $s \in \{\varepsilon, 0, 1\}$ we have that

$$\text{if} \quad ([\check{s}]\overline{H}, [\check{s}]\underline{H}) \vdash Comp(s, P) \xrightarrow[s'u_\varepsilon]{\alpha} (\overline{H}', \underline{H}') \vdash P_1'$$

with u_ε either $[P_u][P_u']$ or $\langle 0s_0s_0'[P_0][P_0'], 1s_1s_1'[P_1][P_1']\rangle$, then

$$(\overline{H}, \underline{H}) \vdash P \xrightarrow[ss'u_\varepsilon]{\alpha'} (\overline{H}'', \underline{H}'') \vdash P' \text{ with } Comp(s, P') = P_1'$$

and α' defined in terms of α, under the following restrictions:

1. for $s = 0$ if $\alpha' = \alpha$ and $(\mathsf{dom}(\overline{H}')\backslash\mathsf{dom}([\check{0}]\overline{H})) \cap fn(Comp(1, P)) = \emptyset$;
2. for $s = 1$ if $\alpha' = \alpha$ and $(\mathsf{dom}(\overline{H}')\backslash\mathsf{dom}([\check{1}]\overline{H})) \cap fn(Comp(0, P)) = \emptyset$;
3. for $s = \varepsilon$ and $P = (\nu n)P_1$ if ($n \notin n(\alpha)$ and $\alpha' = \alpha$) or ($\alpha' = \alpha = \overline{a}\langle\epsilon\rangle n$ and $n \neq a$).

Proof: We take cases after $s \in \{\varepsilon, 0, 1\}$.

1. When $s = 0$ it implies that $P = P_0 \,||\, Q$ since $Comp(0, P) = P_0$, otherwise is undefined or it enters under the ϵ cases below. The requirements for the rule (PAR$_0$) are satisfied by the restrictions of the lemma. Therefore, we have the expected transition $(\overline{H}, \underline{H}) \vdash P_0 \,||\, Q \xrightarrow[0s'u_\varepsilon]{\alpha} (\overline{H}'', \underline{H}'') \vdash P_0' \,||\, Q$ and $Comp(0, P_0' \,||\, Q) = P_0'$, where $(\overline{H}'', \underline{H}'')$ are updated according to the (PAR$_0$) rule, but this is unimportant for this lemma.
2. When $s = 1$ use an argument as for $s = 0$ where we use rule (PAR$_1$) instead.
3. When $s = \varepsilon$ we consider the two non-trivial cases for which $Comp$ is applicable, i.e., when either $P = !P_1$ or $P = (\nu n)P_1$.
   - For $P =!P_1$ we have that $Comp(\epsilon, !P_1) = Comp(\epsilon, P_1 \,||\, !P_1) = P_1 \,||\, !P_1$. Therefore, the transition given by the lemma is in fact $P_1 \,||\, !P_1 \xrightarrow[s'[P_0][P_0']]{\alpha} P_1'$ and we are allowed to apply the rule (REP) to obtain $!P_1 \xrightarrow[s'[P_0][P_0']]{\alpha} P_1'$ which is the transition we are looking for, i.e., having $s = \varepsilon$, $P' = P_1'$, and $Comp(\epsilon, P_1') = P_1'$. The histories do not change, so we omitted them.
   - For $P = (\nu n)P_1$ we have $Comp(\epsilon, (\nu n)P_1) = Comp(\epsilon, P_1) = P_1$. We take two cases:
     - where $n \notin n(\alpha)$ and $\alpha = \alpha'$ for which the only applicable rule is (SCOPE) where we have the transition $([\check{\epsilon}]\overline{H}, [\check{\epsilon}]\underline{H}) \vdash (\nu n)P_1 \xrightarrow[s'[P_1][P_0']]{\alpha} (\overline{H}', \underline{H}') \vdash (\nu n)P_1'$. We also have that $Comp(\epsilon, (\nu n)P_1') = Comp(\epsilon, P_1') = P_1'$ and $([\check{\epsilon}]\overline{H}, [\check{\epsilon}]\underline{H}) = (\overline{H}, \underline{H})$ which end this case.
     - when $\alpha = \overline{a}\langle\epsilon\rangle n$ and $n \neq a$, the only rule that can work with this action label is the (OPEN) rule, which will give us the transition $([\check{\epsilon}]\overline{H}, [\check{\epsilon}]\underline{H}) \vdash (\nu n)P_1 \xrightarrow[s'[P_0][P_0']]{\alpha} (\overline{H}'', \underline{H}'') \vdash P_1'$ and as $Comp(\epsilon, P_1') = P_1'$ and $([\check{\epsilon}]\overline{H}, [\check{\epsilon}]\underline{H}) = (\overline{H}, \underline{H})$ the statement of the lemma is true.

□

Corollary 3.4.10: Applying several times Lemma 3.4.9, when the needed restrictions exist, we can extend s to be a string of location components: $s \in \{0, 1\}^*$.

Lemma 3.4.11 (composing communications): Whenever we have

$$([\check{0}]\overline{H}, [\check{0}]\underline{H}) \vdash Comp(0, P) \xrightarrow[s_0[P_0][P_0']]{\overline{a}\langle\tilde{b}\rangle n} (\overline{H}'_0, \underline{H}'_0) \vdash P_0''$$

and

$$([\check{1}]\overline{H}, [\check{1}]\underline{H}) \vdash Comp(1, P) \xrightarrow[s_1[P_1][P_1']]{\underline{a}(n)} (\overline{H}'_1, \underline{H}'_1) \vdash P_1'',$$

for $a \notin \mathsf{dom}(\overline{H}'_0 \setminus [\check{0}]\overline{H})$, then we have the communication

$$(\overline{H}, \underline{H}) \vdash P \xrightarrow[\langle 0s_0[P_0][P_0'], 1s_1[P_1][P_1']\rangle]{\tau} (\overline{H}', \underline{H}') \vdash P'$$

with $Comp(0, P') = P_0''$ and $Comp(1, P') = P_1''$. The symmetric case was elided.

Proof: The statement of the lemma implies that $Comp$ is applicable to P, both for a label 0 and 1. This means that P has the structure of a parallel composition. Because of the restriction $a \notin \tilde{b}$ we are allowed to use the (COM) rule, which gives us that $P' = (\nu\tilde{b})(P_0'' \,||\, P_1'')$ after the transition. From Definition 3.4.4(4) we have that $Comp(0, (\nu\tilde{b})(P_0'' \,||\, P_1'')) = Comp(s, (P_0'' \,||\, P_1''))$. which by Definition 3.4.4(2) becomes $Comp(0, P_0'' \,||\, P_1'') = P_0''$. Following the same reasoning using first Definition 3.4.4(4) and then Definition 3.4.4(3) we have that $Comp(1, P_0'' \,||\, P_1'') = P_1''$ proving the lemma. The histories are updated by the rule, but this is not relevant for the lemma, thus we denote them just $(\overline{H}', \underline{H}')$. The symmetric version follows the same lines of reasoning. □

Lemma 3.4.12 (localization): For any process P and a location string s, whenever *Comp* is defined, we have:

1. if $(\overline{H}, \underline{H}) \vdash P \xrightarrow[s[P_1][P_1']]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ and $(s, s') \in I_l$ then $Comp(s', P) = Comp(s', P')$,

2. if $(\overline{H}, \underline{H}) \vdash P \xrightarrow[s\langle s_0[P_0][P_0'], s_1[P_1][P_1']\rangle]{\tau} (\overline{H}', \underline{H}') \vdash P'$ and $(ss_0, s') \in I_l$ and $(ss_1, s') \in I_l$ then $Comp(s', P) = Comp(s', P')$.

Proof: We first prove the part 1.

Since $(s, s') \in I_l$ then we can split s and s' into $s = u0l_1$, $s' = u1l_1'$ respectively where $u = lcp(s, s')$ is the largest common prefix, or the symmetric $s = u1l_1, s' = u0l_1$, which can be treated analogous. From Corollary 3.4.5 we have that $Comp(s, P) = Comp(u0l_1, P) = Comp(0l_1, Comp(u, P))$, and denote $Comp(u, P) = P_u$. The decomposition Lemma 3.4.6 (i.e., Corollary 3.4.7 for u a string) allows us to derive the transition:

$$([\check{u}]\overline{H}, [\check{u}]\underline{H}) \vdash P_u = Comp(u, P) \xrightarrow[0l_1[P_\epsilon][P_\epsilon']]{\alpha'} (\overline{H}'', \underline{H}'') \vdash Comp(u, P') = P_u'$$

with $\alpha' = \alpha$ except for the case when (OPEN) rule is applied in the derivation tree, with $\alpha = \overline{a}\langle n\rangle n$, $a \neq n$ and $\alpha' = \overline{a}\langle\epsilon\rangle n$. Because of the location label $0l_1$ it means that $P_u = P_{0'} \,||\, P_{1'}$, and applying Lemma 3.4.6 to the above transition we derive:

$$([\check{0}][\check{u}]\overline{H}, [\check{0}][\check{u}]\underline{H}) \vdash Comp(0, P_u) = P_{0'} \xrightarrow[l_1[P_\epsilon][P_\epsilon']]{\alpha'} (\overline{H}''', \underline{H}''') \vdash P_{0'}' = Comp(0, P_u').$$

Lemma 3.4.6 also ensures that $(\mathsf{dom}(\overline{H}''') \backslash \mathsf{dom}([\check{u0}]\overline{H})) \cap fn(Comp(1, P_u)) = \emptyset$.

Moreover, since $(0, 1) \in I_l$ we can apply the current lemma inductively to the transition with location $0l_1$ to obtain that $Comp(1, P_u) = Comp(1, P_u')$. These (and the (PAR$_0$) rule that is applied to obtain the last transition) imply that $P_u' = P_{0'}' \,||\, P_{1'}$.

To finish the proof, use Corollary 3.4.5 to get $Comp(s', P) = Comp(1l_1', Comp(u, P))$, and we thus need to prove that

$$Comp(1l_1', Comp(u, P)) = Comp(l', Comp(u, P')).$$

We already know that $Comp(u, P) = P_u$ and $Comp(u, P') = P_u'$. Therefore, we need to show

$$Comp(1l_1', P_u) = Comp(1l_1', P_u').$$

Since we already have proven that $Comp(1, P_u) = Comp(1, P_u')$ this part is done.

For the part 2 we can consider ss_0 and s' to be written as $ss_0 = u_0l_0$ respectively $s' = u_0l_0'$, for which the independence says that u_0 is maximal and l_0 start with a 0 and l_0' with a 1 (the symmetric case is analogous). Similarly, for ss_1 and s' their independence implies that $ss_1 = u_1l_1$ and $s' = u_1l_1'$. Since $u_1l_1' = s' = u_0l_0'$ we can identify two cases:

(i). when $u_1 = u_0$ or

(ii). when $u_0 \prec u_1 \vee u_1 \prec u_0$.

Recall that since they are part of a communication location label the s_0 starts with a 0 and s_1 starts with a 1. The case (i) means that $u_0 \prec s \wedge u_1 \prec s$. Because of this, the component identified by s' is outside any of the two components that participate in the communication, and thus the rest of this case follows as for part 1.

For the case (ii) we know that ss_0 can be written as $s0s_0'$ and ss_1 can be written $s1s_1'$, which implies that $u_0 \prec u_1 \vee u_1 \prec u_0$. We work with the first sub-string inclusion, as the second one would be analogous. In this case we deduce that $u_0 = s$, for otherwise we would break the requirement that u_0 is maximal. This implies that s' can now be written as $s' = s1l_0''$ (when u_1 is the shortest then $s' = s0l_1''$), which means that the component we are working with is in the right part of the communication. Moreover, u_1 can be written as $s1v'$ and thus the whole $s' = s1v'l_1'$ and $ss_1 = s1v'l_1$. Since $s'I_lss_1$ and $s1v'$ is maximal then $l_1'I_ll_1$ (and they start one with 0 and the other with 1).

Using the decomposition Lemma 3.4.8 we obtain the transition

$$([\check{s1v'}]\overline{H}, [\check{s1v'}]\underline{H}) \vdash Comp(s1v', P) \xrightarrow[l_1[R][R']]{\alpha} (\overline{H}''', \underline{H}''') \vdash Comp(s1v')P'.$$

We can apply the current lemma inductively to this transition and the independent labels $l'_1 I_l l_1$ to obtain

$$Comp(l'_1, Comp(s1v', P)) = Comp(l'_1, Comp(s1v')P')$$

which by Corollary 3.4.5 transforms in the expected result. □

Proof of Theorem 3.4.2: It is easy to see that the independence relation I of Definition 3.2.14 is irreflexive, because it inherits this from I_{CCS}, and symmetric, because I_{CCS} is and the second part is symmetric by definition.

For the first *ATS requirement 3.4.1(1)* each event in the generated transition system is a split event, which by Definition 3.2.12 is attached to a transition between two process.

The *ATS requirement 3.4.1(2)* is covered by Lemma 3.4.3.

To prove the *ATS requirement 3.4.1(3)* consider a history $(\overline{H}, \underline{H})$ and some process P, and two events $e = (\alpha, u, D), e' = (\alpha', u', D')$ enabled in this state $(\overline{H}, \underline{H}) \vdash P$, which are also independent eIe'. Take the two transitions corresponding to these events in the generated transition system, i.e.:

$$(\overline{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\overline{H}', \underline{H}') \vdash P' \quad \text{and} \quad (\overline{H}, \underline{H}) \vdash P \xrightarrow[u',D']{\alpha'} (\overline{H}'', \underline{H}'') \vdash P''.$$

To satisfy 3.4.1(3) we need to prove the existence of the following transitions:

$$(\overline{H}', \underline{H}') \vdash P' \xrightarrow[u',D']{\alpha'} (\overline{H}''', \underline{H}''') \vdash P''' \quad \text{and} \quad (\overline{H}'', \underline{H}'') \vdash P'' \xrightarrow[u,D]{\alpha} (\overline{H}'''', \underline{H}'''') \vdash P''''$$

with $H''' = H'''', P''' = P''''$.

We can have three possible combinations of events, depending on the structure of their location labels:

1. $u = s[R_0][R'_0]$ and $u' = s'[R_1][R'_1]$;
2. $u = s\langle s_0[R_0][R'_0], s_1[R_1][R'_1]\rangle$ and $u' = s'[R_2][R'_2]$;
3. $u = s\langle s_0[R_0][R'_0], s_1[R_1][R'_1]\rangle$ and $u = s'\langle s_2[R_2][R'_2], s_3[R_3][R'_3]\rangle$.

We treat the first case.

Since eIe' then Definition 3.2.14 implies that $eI_{CCS}\,e'$ and that $\nexists n : D(n) = u' \wedge \nexists n : D'(n) = u$ (which we use towards the end of the proof, when showing the ATS property 3.4.1(4)). Since $Loc(e) = \{s\}$ and $Loc(e') = \{s'\}$ we have by Definition 3.2.7 that $sI_l s'$ which means that they look like $s = v0s_3$ and $s' = v1s'_3$ with $v = lcp(s, s')$ the largest common prefix. In consequence, $Comp(v, P) = P_0 \,||\, P_1$ is a parallel composition, and the decomposition Lemma 3.4.6 applied using v allows to derive the transitions

$$([\check{v}]\overline{H}, [\check{v}]\underline{H}) \vdash Comp(v, P) \xrightarrow[0s_3[R_0][R'_0]]{\alpha_0} (\overline{H}'_v, \underline{H}'_v) \vdash Comp(v, P') \quad \text{and}$$

$$([\check{v}]\overline{H}, [\check{v}]\underline{H}) \vdash Comp(v, P) \xrightarrow[1s'_3[R_1][R'_1]]{\alpha'_0} (\overline{H}''_v, \underline{H}''_v) \vdash Comp(v, P'').$$

Note that α_0 is not necessarily the same as the α (cf. Lemma 3.4.6). For this part of the proof we can ignore the D, D' sets because these are derived from the histories of transitions like the above. Therefore, we reason over simpler transitions as above, and in the end we argue that the same split transitions on the respective D and D' can be obtained from the resulting histories.

At this point the two derivation trees are different in the sense that in one the (PAR$_0$) rule is applicable whereas in the other the (PAR$_1$) rule. From the two (PAR) rules we have the following respective transitions

$$([\check{v0}]\overline{H}, [\check{v0}]\underline{H}) \vdash P_0 \xrightarrow[s_3[R_0][R'_0]]{\alpha_0} (\overline{H}'_0, \underline{H}'_0) \vdash P'_0 \quad \text{and} \quad ([\check{v1}]\overline{H}, [\check{v1}]\underline{H}) \vdash P_1 \xrightarrow[s'_3[R_1][R'_1]]{\alpha'_0} (\overline{H}'_1, \underline{H}'_1) \vdash P'_1.$$

According to the decomposition Lemma 3.4.6 we also have $Comp(v0, P) = P_0$ and $Comp(v1, P) = P_1$, as well as $Comp(v0, P') = P'_0$ and $Comp(v1, P'') = P'_1$. Moreover, Lemma 3.4.6 also provides the following $\mathsf{dom}(\overline{H}'_0) \setminus \mathsf{dom}([\check{v0}]\overline{H}) \cap fn(P_1) = \emptyset$ respectively $\mathsf{dom}(\overline{H}'_1) \setminus \mathsf{dom}([\check{v1}]\overline{H}) \cap fn(P_0) = \emptyset$.

Now we show how to derive the first expected transition $(\overline{H}', \underline{H}') \vdash P' \xrightarrow[u',D']{\alpha'} (\overline{H}''', \underline{H}''') \vdash P'''$. Because of the localization Lemma 3.4.12 applied to labels $0s_3I_l1$ we know that $P_1 = Comp(1, Comp(v, P)) = Comp(1, Comp(v, P'))$ and thus we can deduce that $Comp(v, P') = P'_0 \,||\, P_1$. To this we can now apply the rule (PAR$_1$) with the right transition from above, and deduce the transition

$$(\overline{H}'_v, \underline{H}'_v) \vdash P'_0 \,||\, P_1 = Comp(v, P') \xrightarrow[1s'_3[R_1][R'_1]]{\alpha'_0} (\overline{H}'''_v, \underline{H}'''_v) \vdash P'_0 \,||\, P'_1.$$

We were able to apply the right transition from above with a different (correct) history because of Lemma 3.3.4. Moreover, the requirement of the (PAR$_1$) rule (i.e., $\tilde{b} \cap fn(P'_0) = \emptyset$, with $\tilde{b}$ the change in histories from the respective transition above) was provided above by Lemma 3.4.6 but through the result in Corollary 3.3.5, which says that the change of histories $\tilde{b}$ is the same for any starting histories (using moreover the fact that $fn(P_0) = fn(P'_0)$).

In order to obtain the full required transition we need to show that now we can apply the Lemma 3.4.9 for composing transitions on the above last transition (i.e., the respective restrictions of Lemma 3.4.9 need to be satisfied). For this we consider a minimal $v \in \{0, 1\}$ and take cases after it.

Consider $v = 0$ (or for $v = 1$ an analogous argument will go through), i.e., we use $s = 00s_3$ and $s' = 01s'_3$. Apply the localization Lemma 3.4.12 with the independent locations $s = 00s_3I_l1$ to deduce that $Comp(1, P) = Comp(1, P')$, and therefore, $P' = P'_0 \,||\, P_1 \,||\, Q_1$.

In order to apply the composition Lemma 3.4.9 we must satisfy the corresponding restriction, i.e.: $(\mathsf{dom}(\overline{H}'_0) \setminus \mathsf{dom}([\check{0}]\overline{H})) \cap fn(Comp(1, P') = Q_1)$ and that $\alpha'_0 = \alpha'$. These two are given by the decomposition Lemma 3.4.6 when it was applied to the second transition with the above particular $v = 0$. This is because the lemma had to apply the (PAR$_0$) rule which keeps the same action names, i.e., $\alpha' = \alpha'_0$ and the restriction on names $(\mathsf{dom}(\overline{H}'_0) \setminus \mathsf{dom}([\check{0}]\overline{H})) \cap fn(Comp(1, P))$ which is equal, as mentioned above, to Q_1.

To any of the above we can apply rules that do not change the location label, i.e. the (OPEN), (SCOPE), or (REP). The first two need the form of P to be $P = (\nu n)Q$, for which $Comp(v, P) = Q$ for any location label v. The change of histories will tell which of (OPEN) or (SCOPE) have been applied. For any of the two rules the decomposition Lemma 3.4.6 property 3 ensures that $\alpha'_0 = \alpha'$. Moreover, the restrictions needed by the composition Lemma 3.4.9(3) are provided.

In consequence, in all cases we can apply the composition Lemma 3.4.9 to $Comp(v, P') \xrightarrow[1s'_3[R_1][R'_1]]{\alpha'_0} P'_0 \,||\, P'_1$ to obtain $P' \xrightarrow[v1s'_3[R_1][R'_1]]{\alpha'_0} P'''$ with $Comp(v, P''') = P'_0 \,||\, P'_1$.

We can use analogous reasoning to obtain a transition $Comp(v, P'') \xrightarrow[0s_3[R_0][R'_0]]{\alpha_0} P'_0 \,||\, P'_1$. Again, $\alpha = \alpha_0$ by the decomposition Lemma 3.4.6(3). We can apply the composition Lemma 3.4.9 to obtain $P'' \xrightarrow[v0s_3[R_0][R'_0]]{\alpha_0} P''''$ with $Comp(v, P'''') = P'_0 \,||\, P'_1$.

It remains to argue that $P''' = P''''$. This is the case because of the localization Lemma 3.4.12 which says that for any independent location label $v'I_lv$ we have:

- $Comp(v', P) = Comp(v', P')$ as well as $Comp(v', P) = Comp(v', P'')$ from the given transitions;
- $Comp(v', P') = Comp(v', P''')$ as well as $Comp(v', P'') = Comp(v', P'''')$ for the above deduced transitions.

Therefore, we get $Comp(v', P''') = Comp(v', P'''')$, which together with the fact that $Comp(v, P''') = P'_0 \,||\, P'_1 = Comp(v, P'''')$ we have our expected result $P''' = P''''$.

Claim: The histories H' and H'' are changed by the two derived transitions into the same history $H''' = H''''$.

We look in the derivation tree of the transition to identify how histories are being changed. Observe that rules (SUM), (REP), (SCOPE), copy the histories from the hypothesis to the conclusion, i.e., they only propagate down the tree the changes done further up by the other rules. Rule (OUT) does no change to the histories.

We discuss how the rest of the rules (IN), (PAR_i), (OPEN), (COM_i), change the histories.

The only rule that adds new names to the input history $\underline{H}$ is (IN) and this can only be applied once in a derivation tree. Therefore, the action label α of the transition will tell whether the (IN) rule has been applied, and thus we know exactly what has been added to the input history, i.e., the pair (n, ε) with n the name from the action $\alpha = \underline{a}(n)$. Moreover, when (IN) is applied then in the same derivation tree the rule (OUT) cannot appear, and thus also not the rule (OPEN) as it needs further up in the tree an application of (OUT) (as seen from the action label in the hypothesis of (OPEN)). In consequence, in a derivation tree, together with rule (IN) only (PAR) or (COM) rules may still be applicable. The (PAR) rules update any history information coming from further up in the derivation tree by adding the respective component label, i.e., replacing $[i]\overline{H}$ by $i\overline{H}'_i$ for extruder histories and $[i]\underline{H}$ by $i\underline{H}'_i$ for input histories. The (COM) rules update the input histories with the label of the component that does the input in the communication.

The (COM) rules, on the other hand, do not change the extruder histories (the same as e.g., (OUT) does not), since in the conclusion we have the same output histories both on the left and on the right of the transition. But further up in the derivation tree histories may change, in response to (OPEN) or (PAR_i).

The (PAR) rules add new information to the extruder history as $\overline{H}''$. The pair that is added in the $\overline{H}''$ depends on the action label $\alpha = \overline{a}\langle\tilde{b}\rangle n$ and on the history that we work with (both input and extruder parts of the history). In any case, at most one name is added to the history. Moreover, when (IN) is applicable, then $\overline{H}''$ is empty because it depends on α being an output action.

All these tell that when the actions on the transitions that we work with, i.e., either α or α' are input actions, then the history is changed by adding one name pair to the input history $\underline{H}$ and leaving the output history unchanged.

The (OPEN) rule requires a previous application of (OUT), and therefore, no application of (IN) is possible, thus the input history $\underline{H}$ remains unchanged throughout the derivation tree. The (OPEN) rule adds one single pair (n, u) which we extract from the transition labels, i.e., having $\alpha = \overline{a}\langle b\rangle n$ and the location label u. Note that at this point in the derivation tree the name n is added as extruder with the current location u, but this location is updated by the (PAR) rules through prefixing with location labels depending on the respective component. Thus, at the root of the tree this extruded name appears as new in the history but with the full label u which we see on the transition that we work with.

To finish the proof of this claim we take cases after α and α'.

1. When both $\alpha = \underline{a}(n)$ and $\alpha' = \underline{b}(m)$ are input actions.

   The given histories are $H' = H \cup H_1$ where $H_1 = (\emptyset, \{(n, u)\})$, and $H'' = H \cup H_2$ where $H_2 = (\emptyset, \{(m, u')\})$. Important is that adding new entries to the input histories does not depend on the previous history H. Therefore, when we apply the input actions in the derived transitions, i.e., α' to H' and α to H'', we add the respective input names. This means that $H''' = H' \cup H_2 = H \cup H_1 \cup H_2$ and $H'''' = H'' \cup H_1 = H \cup H_2 \cup H_1$, which are the same. This reasoning works even when some of the names a, x, n, b, y, m are equal.

2. When both $\alpha = \overline{a}\langle\tilde{n}\rangle n$ and $\alpha' = \overline{b}\langle\tilde{m}\rangle m$ are output actions, where both n, m can be extruded names or not, and not necessarily different.

   (a) Consider both actions extrude their name and $n \neq m$; then the given histories are $H' = H \cup H_1$ where $H_1 = (\{(n, u)\}, \emptyset)$, and $H'' = H \cup H_2$ where $H_2 = (\{(m, u')\}, \emptyset)$. Since both actions extrude their names, it means that the (OPEN) rule has been applied in both derivation trees. Whenever (OPEN) is applied then we are guaranteed that the name added to the extruder history does not already exist in the extruder history part, i.e., (OPEN) cannot be applied two times with the same name during the execution of a process.

   Therefore, if H_1 adds name n to H and H_2 adds name m, and neither of the names existed in H, then it is obvious that when in the deduced transitions the α' is applied to H' it will add H_2. The same for the other transition to obtain the expected result as in the previous case.

   (b) The case when only one of the actions extrudes the name $n \neq m$ is handled by the (PAR) rules. For the (PAR) rule to add to the extruder history, i.e., so $\overline{H}'' \neq \emptyset$, the name that needs to be added must already exist in the history but with an independent location label. Assume this for α, thus $(n, u'') \in \overline{H}$ with $u'' I_l u$. When α' adds a different name m then $H'' = H \cup H_2$ where $H_2 = (\{(m, u')\}, \emptyset)$ will also have

$(n,u'') \in \overline{H}''$. Therefore α will still add the (n,u), and thus we get the same histories in both deduced transitions.

(c) When $n = m$ and any of α or α' can contain ϵ, then we are in the situation of parallel extruder's. Here is the case when in one transition we apply the rule (OPEN) and in the other the rule (PAR). This is fine since the name n will be initially added to the histories H' and H'' by the (OPEN) rule, and then in the deduced transitions it will satisfy the requirements to be added to the histories again, but with an independent location.

3. When α is an input action and α' is an output action the reasoning is similar to the one done in the first case, but working with different rules.

Claim: The deduced transitions can be split with the respective D, D' based on the respective histories.

From the two given transitions we know that both $D \subseteq \overline{H}$ and $D' \subseteq \overline{H}$. Since the histories only (at most) increase through transitions and we have seen that $H \subseteq H''$ then we also have that $D \subseteq H''$, meaning that the transition can be split with D (the same for $D' \subseteq H'$).

For the second case, when locations $u = s\langle s_0[R_0][R_0'], s_1[R_1][R_1']\rangle$ and $u' = s'[R_2][R_2']$, the proof follows similar arguments as for the first case, with few differences which we comment about in the following. Note that the independence of the events now offers two pairs of independent locations: $ss_0 I_l s'$ and $ss_1 I_l s'$, which give rise to several cases. Recall that s_0 and s_1 start with a 0 respectively 1.

Assume that in both independences we have $v = lcp(ss_0, s') \prec s$ and $v' = lcp(ss_1, s') \prec s$, which means that $v = v'$. It is easy to see that now we can immediately apply the same reasoning as we did before, for the first case. Moreover, recall that the (COM) rule which is applied for the transition with u does not change histories, and thus this transition does not change histories.

Assume that $v = s$ which means that $ss_0 = v0s_0'$ and $s' = v1s_3'$. This in turn implies that $s_3' \neq \varepsilon$ because otherwise we would get $s' \prec ss_1 = v1s_1'$ which breaks the independence of these two. Moreover, since $v1s_1' = ss_1 I_l s' = v1s_3'$ we get that $s_1' I_l s_3'$ and denote $lcp(s_1', s_3') = v''$, thus having $ss_1 = v1v''0s_1''$ and $s' = v1v''1s_3''$ (or a symmetric variant).

Because of the form of the location u we can use the fact that to create a communication location label the (COM) rule must have been applied in the derivation tree, which requires that $Comp(v,P) = P_0 \,||\, P_1$, thus, having the following transitions

$$H, P \xrightarrow[v\langle 0s_0'[R_0][R_0'], 1v''0s_1''[R_1][R_1']\rangle]{[\tau],L} H, P' \quad \text{and} \quad H, P \xrightarrow[v1v''1s_3''[R_2][R_2']]{[\alpha'],L'} H'', P''.$$

We must deduce the existence of the following two transitions

$$H, P' \xrightarrow[v1v''1s_3''[R_2][R_2']]{[\alpha'],L'} H''', P''' \quad \text{and} \quad H'', P'' \xrightarrow[v\langle 0s_0'[R_0][R_0'], 1v''0s_1''[R_1][R_1']\rangle]{[\tau],L} H'''', P'''',$$

with $H''' = H''''$ and $P''' = P''''$. We concentrate on deducing the left transition.

Using Lemma 3.4.6 on the first given transition we can find the transition

$$Comp(v,P) \xrightarrow[\langle 0s_0'[R_0][R_0'], 1v''0s_1''[R_1][R_1']\rangle]{[\tau],L} Comp(v,P')$$

which will be from $P_0 \,||\, P_1$ to some $P_0' \,||\, P_1'$. Using the decomposition of communications Lemma 3.4.8 we get two transitions

$$Comp(0, Comp(v,P)) \xrightarrow[s_0'[R_0][R_0']]{\alpha} Comp(0, Comp(v,P')) \text{ and}$$

$$Comp(1, Comp(v,P)) \xrightarrow[v''0s_1''[R_1][R_1']]{\overline{\alpha}} Comp(1, Comp(v,P')).$$

Applying more the decomposition Lemma 3.4.6 and Corollary 3.4.5 to this last transition we obtain

$$Comp(v1v'', P) \xrightarrow[0s_1''[R_1][R_1']]{\overline{\alpha}} Comp(v1v'', P').$$

Whereas applying the same decomposition Lemma 3.4.6 to the second given transition we obtain

$$Comp(v1v'', P) \xrightarrow[1s''_3[R_2][R'_2]]{[\alpha'],L'} Comp(v1v'', P'').$$

Because of the localization Lemma 3.4.12 and the independence of these last two location labels we can see that we can apply the same transition with α' but to the component of P', i.e., deduce

$$Comp(v1v'', P') \xrightarrow[1s''_3[R_2][R'_2]]{[\alpha'],L'} Comp(v1v'', P'').$$

We can lift this transition using localization Lemma 3.4.12 and the composition of transitions Lemma 3.4.9 because the restrictions are satisfied by the previous decomposition lemma applications (similarly as we argued in the first case above) and obtain the transition

$$Comp(v, P') \xrightarrow[1v''1s''_3[R_2][R'_2]]{[\alpha'],L'} Comp(v, P'').$$

Because of the independence with the label $0s'_0$ we can apply the localization lemma to deduce that this part of the process also remains unchanged. Therefore, when applying again the composition Lemma 3.4.9 we lift the transition to the top most process, as expected

$$P' \xrightarrow[v1v''1s''_3[R_2][R'_2]]{[\alpha'],L'} P'''$$

with P''' the same as P' only with the component $Comp(v1v'', P')$ changed accordingly. This component is different than the part that was changed through the communication.

To deduce the second transition apply a similar argument, but when lifting up apply the composition of communications Lemma 3.4.11 to deduce the correct τ-transition. The resulting process will be the same as P''' because when first we applied the transition with α' we change a part which was then not touched by the communication transition.

It is easy to see that these deduced transitions are allowed, and that the histories are the same, i.e., $H''' = H'''' = H''$.

For the fourth ATS requirement of Definition 3.4.1(4) consider a history H, some process P, an event $e = (\alpha, u, L)$ with a transition $H, P \xrightarrow[u]{[\alpha],L} H', P'$, and an event $e' = (\alpha', u', L')$ with a transition $H', P' \xrightarrow[u']{[\alpha'],L} H'', P''$, where we also have that eIe'. To satisfy 3.4.1(4) we need to prove the existence of the following transitions

$$H, P \xrightarrow[u']{[\alpha'],L'} H''', P''' \quad \text{and} \quad H''', P''' \xrightarrow[u]{[\alpha],L} H'', P''.$$

We will only prove for the first of the transitions and rely on the proof for 3.4.1(3) to show the existence of the second transition. This is the case because once we have deduced the transition from P, deducing the fourth transition will fall in under the 3.4.1(3).

We have four different combinations of events depending on the structure of their location labels:

1. $u = s[R_0][R'_0]$ and $u' = s'[R_1][R'_1]$;
2. $u = s\langle s_0[R_0][R'_0], s_1[R_1][R'_1]\rangle$ and $u' = s'[R_2][R'_2]$;
3. $u = s[R_0][R'_0]$ and $u' = s'\langle s_1[R_1][R'_1], s_2[R_2][R'_2]\rangle$;
4. $u = s\langle s_0[R_0][R'_0], s_1[R_1][R'_1]\rangle$ and $u = s'\langle s_2[R_2][R'_2], s_3[R_3][R'_3]\rangle$.

For case 1 we first write s and s' as we did in the proof of case 1 for 3.4.1(3) as $s = lcp(s, s')0v$ and $s' = lcp(s, s')1v'$ (the symmetric versions follows just the same). Using the decomposition Lemma 3.4.6 on the two given transitions we have the two transition

$$Comp(lcp(s, s'), P) \xrightarrow[0v]{\alpha} Comp(lcp(s, s'), P')$$

and, coming after it,

$$Comp(lcp(s,s'),P') \xrightarrow[1v']{\alpha'} Comp(lcp(s,s'),P'').$$

We want to prove the existence of the transition $Comp(lcp(s,s'),P) \xrightarrow[1v']{\alpha'} P_0'''$. Having this we can employ the composition Lemma 3.4.9, because its restrictions are offered by the previous decomposition lemma, similar to what we argued before, and we deduce the required transition $P \xrightarrow[lcp(s,s')1v']{\alpha'} P'''$ with $Comp(lcp(s,s'),P''') = P_0'''$.

Since $0vI_l1v'$ we can apply the localization Lemma 3.4.12 to the first transition to deduce that

$$Comp(1v', Comp(lcp(s,s'),P)) = Comp(1v', Comp(lcp(s,s'),P')).$$

In consequence we can apply the second transition to this component to obtain the transition that we are looking for, with $P_0''' = Comp(lcp(s,s'),P'')$. Here we have to apply the decomposition and composition lemmas to go down respectively up the location string, similar to what we argued in the previous relevant case for ATS restriction 3.4.1(3).

It remains to make sure that the above derived transition is allowed, which by Definition 3.3.1 means to show that $L' \subseteq \overline{H}$. We know that $L' \subseteq \overline{H}'$ and we know that $H' = H \cup H_0$ where H_0 may be of the form $(\{n, u|_{\{0,1,[\mathbf{P}]\}}\}, \emptyset)$. In fact, any name that is added to $\overline{H}_0$ by α will have the location $u|_{\{0,1,[\mathbf{P}]\}}$. To show our inclusion we can show that $L' \cap \overline{H}_0 = \emptyset$. This is given by the independence relation $eI_\pi e'$ which implies that $\nexists n : L'(n) = u|_{\{0,1,[\mathbf{P}]\}}$.

The remaining three cases follow similar arguments as the above case 1 and the respective cases 2 and 3 from the proof for the ATS requirement 3.4.1(3). □

3.5 Conclusion and Related work

We provided the first stable, non-interleaving operational semantics for the pi-calculus conservatively generalising the interleaving early operational semantics. The semantics is given as labelled asynchronous transition systems. We followed and conservatively generalised the approach for CCS in [MN92] by capturing the link causalities introduced in the pi-calculus processes by employing a notion of extrusion histories.

We are currently working on a more thorough comparison with the related work [MP95, JJ95, BG95, BS95, San96, CS00, CVY12, CKV13, Cri15], in particular we aim to explore the differences between early and late style non-interleaving semantics. Finally, we work on extending this work to the Psi-calculus [BJPV11].

4. TOWARDS NON-INTERLEAVING SEMANTICS FOR PSI-CALCULI

In this chapter we look at the state of developing non-interleaving semantics for Psi-calculus. We look at the issues that has to be solved for a non-interleaving semantic for Psi-calculi to be possible. The work in this chapter is mostly on the level of discussions and suggestions, rather than that of completed results. We do try to give as good an idea as possible where we plan to take this, how we think a solution would look like and in the cases we have, give initial proof sketches of results. The main issue that we try to identify in this chapter is the notion of causality introduced by the assertions in Psi-calculi, which we name environmental causality. Other possible names for this is logical causality or conditional causality. We will look more into this in the next Section where we identify the causalities that appear in Psi-calculi.

4.1 Causality in Psi-calculi

In order to give non-interleaving semantics to Psi-calculi we have to know what dependencies that can arise. We also need to know how to, in a semantic, discover what these dependencies are, and give an independence relation that encompass all the dependencies we find.

Psi-calculi inherit both the structural and the linked causality from pi-calculus. This can be seen as Psi-calculi have the same form of prefixing and parallel constructs as is found in pi-calculus, this is what gives the structural causalities. Psi-calculi also have name-passing as in pi-calculus giving the linked-causalities. While it shares linked causality with pi-calculus there is a major difference: Where pi only allows single names to be used as channel and message does Psi allow for any nominal term to be used as channel and message. While this does not change the way linked causality work, it changes how much impact linked causality has in Psi compared to pi. Where a single transition in pi could only be dependent on at most two extruder's, it is not possible to give a fixed bound for Psi on how many other transitions a single transition can be link-dependent on. In addition to inheriting both the structural and linked causality from pi, Psi-calculi also introduce a new notion of causality. The new notion of causality comes from the enabling function between assertions and conditions, and from how some actions changes the assertions causing conditions to be enabled or disabled, as seen in Example 4.1.1. This new notion of causality and be called the environmental, logical or conditional causality. We will use the name environmental causality for the rest of this thesis.

Example 4.1.1: Take a Psi-calculi instance with the following parameters:

$$\mathbf{T} \stackrel{def}{=} \{a, b, c, d, e\}$$

$$\mathbf{C} \stackrel{def}{=} \mathbf{A} \stackrel{def}{=} \mathbb{Z}$$

$$\mathbf{1} \stackrel{def}{=} 0$$

$$\Psi_a \otimes \Psi_b \stackrel{def}{=} \Psi_a + \Psi_b$$

$$a \dot{\leftrightarrow} b \stackrel{def}{=} a = b$$

$$\Psi \vdash \varphi \stackrel{def}{=} \varphi \leq \Psi$$

Here we have that assertions and conditions are integers, and a condition is enabled only if it is equal to or smaller than the assertion. Channel-equivalence is if the channels are exactly the same, assertion composition is addition of integers making unit to be 0.

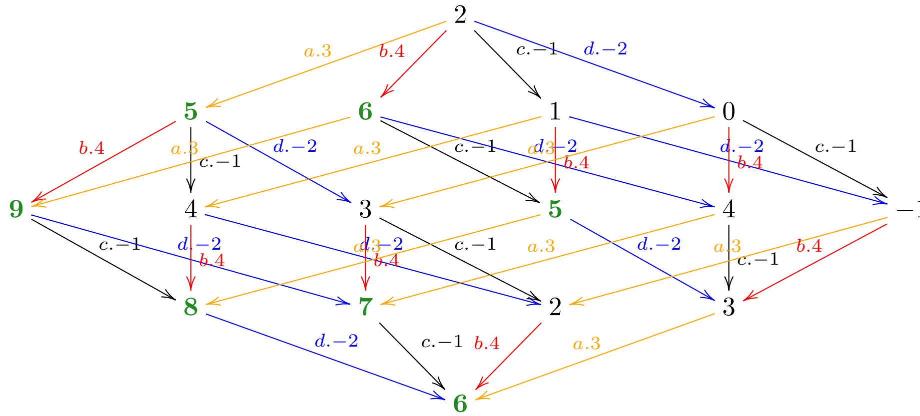


Fig. 4.1: Simplified partial transition system of the process:
$P = (\!|2|\!) \,||\, \overline{a}\langle a\rangle.(\!|3|\!) \,||\, \overline{b}\langle b\rangle.(\!|4|\!) \,||\, \overline{c}\langle c\rangle.(\!|-1|\!) \,||\, \overline{d}\langle d\rangle.(\!|-2|\!) \,||\, \textbf{case } 5 : \overline{e}\langle e\rangle.(\!|-5|\!)$.
The states shown are the environment for the process that would be at that place in the transition system. The transitions are marked with the channel the transition is over and the change the transition does to the environment. The transitions for $\overline{e}\langle e\rangle$ has been omited and it is marked with boldfaced green states (those states that have numeric value of 5 or higher) where $\overline{e}\langle e\rangle$ is enabled and can happen.

This instance can be used to model a very simplified version of someone's financial status. The current environment gives how much a person is worth in a monetary view, be that as fortune if the environment is positive or as debt if it is negative. The actions can be viewed as financial transactions, be that moving money from one account to another, receiving payment or paying bills among other things. The actions may leave behind new assertions that can be viewed as earnings coming in or expenditure depending on if the assertion is positive or negative. A guard is blocking some transactions that demands a certain wealth (or not to much debt) in order to be taken. One example from the Norwegian laws as of this writing is that one cannot buy property without being able to pay 15% of the price up front. Another example is that you cannot take more loans (or should not be able to) if you have too much debt already.

In this instantiation let us create the following process:

$$P = (\!|2|\!) \,||\, \overline{a}\langle a\rangle.(\!|3|\!) \,||\, \overline{b}\langle b\rangle.(\!|4|\!) \,||\, \overline{c}\langle c\rangle.(\!|-1|\!) \,||\, \overline{d}\langle d\rangle.(\!|-2|\!) \,||\, \textbf{case } 5 : \overline{e}\langle e\rangle.(\!|-5|\!)$$

We generate a simplified partial transition system from P in Figure 4.1 where the states are shown only by the environment that exist for that state, the transitions for the actions $\overline{a}\langle a\rangle$, $\overline{b}\langle b\rangle$, $\overline{c}\langle c\rangle$ and $\overline{d}\langle d\rangle$ are labelled by the channel name used in the action and how this action affects the environment. Each channel is given a different color in order to make the system more readable. We omit the transition for action $\overline{e}\langle e\rangle$ and show in which states it is enabled by marking them as boldface green (they can also be seen by the numeric value being 5 or greater).

In this we can see that getting to a state enabling the action $\overline{e}\langle e\rangle$, is dependent on which of the other actions we have had so far. Initially we need either $\overline{a}\langle a\rangle$ or $\overline{b}\langle b\rangle$ to be executed for $\overline{e}\langle e\rangle$ to be enabled. But if $\overline{c}\langle c\rangle$ happens first then $\overline{b}\langle b\rangle$ must happen for $\overline{e}\langle e\rangle$ to be enabled, if $\overline{d}\langle d\rangle$ happens then both $\overline{a}\langle a\rangle$ and $\overline{b}\langle b\rangle$ are needed to enable $\overline{e}\langle e\rangle$. If we on the other hand have the action $\overline{b}\langle b\rangle$ before the action $\overline{c}\langle c\rangle$ then $\overline{c}\langle c\rangle$ will have no impact on whether $\overline{e}\langle e\rangle$ is enabled or not.

The example above shows that a single action may or may not enable/disable other actions depending on the environment. However, it is also possible in Psi that a single action may enable some other action in one environment, disable the same action in another and have no effect in a third. This can be seen in the example bellow.

Example 4.1.2: Take a Psi-calculi instance with the following parameters:

$$\mathbf{T} \overset{def}{=} \{a, b, c, d, e\}$$

$$\mathbf{A} \overset{def}{=} \mathbb{Z}$$
$$\mathbf{C} \overset{def}{=} \mathbb{Z} \times \mathbb{Z}$$
$$\mathbf{1} \overset{def}{=} 0$$
$$\Psi_a \otimes \Psi_b \overset{def}{=} \Psi_a + \Psi_b$$
$$a \leftrightarrow b \overset{def}{=} a = b$$
$$(a, b) \vdash c \overset{def}{=} a \leq c \wedge c \leq b$$

Here we have the same terms, assertions, composition channel equivalence and unit as in example 4.1.1. Conditions are intervals in which the assertions has to fit in order to be entailed.

In this instance assume we have the following process:

$$P' = (\!|\Psi|\!) \,||\, \overline{a}\langle a\rangle.(\!|3|\!) \,||\, \mathbf{case}\ (4,7)\ : \overline{b}\langle b\rangle.\mathbf{0}$$

If and how $\overline{a}\langle a\rangle$ enables, disables or are arguably independent of $\overline{b}\langle b\rangle$, depends on what Ψ is in the following ways:

- If Ψ is 0 or less or 8 and higher, then $\overline{b}\langle b\rangle$ is not enabled and can not be enabled by $\overline{a}\langle a\rangle$.
- If Ψ is in the range 1 to 3 then $\overline{b}\langle b\rangle$ becomes enabled after $\overline{a}\langle a\rangle$ happens.
- If Ψ is 4 then $\overline{b}\langle b\rangle$ is enabled regardless to if $\overline{a}\langle a\rangle$ has happened or not.
- If Ψ is in the range 5 to 7 then $\overline{b}\langle b\rangle$ is enabled until $\overline{a}\langle a\rangle$ happens.

As Ψ can be looked at as the environment of this small process, we see that a single action may enable, disable or have no effect on another action depending on what the environment is at that time.

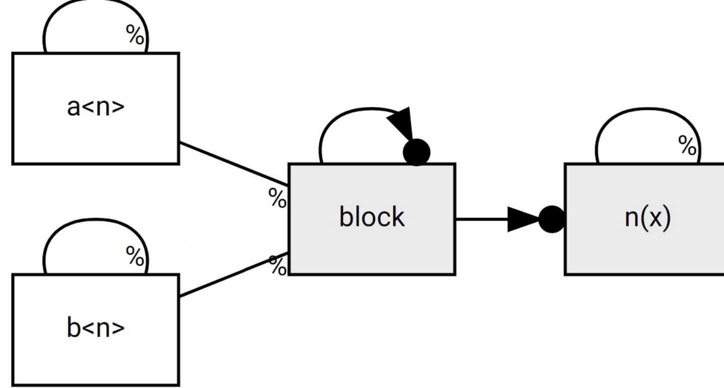
In Example 4.1.1 we see that after the action $\overline{b}\langle b\rangle$, the action $\overline{c}\langle c\rangle$ is no longer able to change the enabling state of $\overline{e}\langle e\rangle$. We are clearly unable to claim that for the initial process and environment that $\overline{c}\langle c\rangle$ is not independent of $\overline{e}\langle e\rangle$ as it does disable $\overline{e}\langle e\rangle$ in the case where only $\overline{a}\langle a\rangle$ has happened. A question that needs to be answered is: Does $\overline{c}\langle c\rangle$ become independent of $\overline{e}\langle e\rangle$ once $\overline{b}\langle b\rangle$ happens? or more general: can an event or action become independent of another due to changes in the environment, even though they are not independent initially?

If we would answer yes, would that mean that the notion of independence between actions are environment dependent? and can changes to the environment make events no longer independent? Just as we claimed they may become independent from a changing environment. If two actions for some environment are both enabled, and if either would be executed the other would not be disabled, can we then not argue for them being able to happen at the same time?

In order to answer this question we look at [DHS15b], where they explore the notion of concurrency and give an independence relation for DCR-graphs. DCR-graphs have the property that one event may enable or disable another event depending on the marking, and then previously disabled events may become enabled again at a later stage. This enabling and disabling of events/actions behaves much the same way as we observe from the environmental causality in Psi-calculi. We can also claim that the causality of DCR-graphs are environmental due to the marking directly affecting enabling of events. One may argue that the relations of DCR-graphs are of a structural nature and thus should give structural dependencies, and we are able to use similar arguments about the case constructs in Psi-calculi. As the relations of DCR-graphs determine which markings enable some event, will the conditions of a case construct determine which assertions that enables the transition or transitions that condition is a guard for.

In [DHS15b], the argument is that one should always be able to swap two independent events that are adjacent in an execution trace. This is the same as claiming that independence must be defined for all environments. While this is a valid argument it is directly opposed to the argument made in [CVY12] on independence between multiple extruders in linked causality. This is particular clear in the pi-calculus process $P = (\nu n)(\overline{a}\langle n\rangle \,||\, \overline{b}\langle n\rangle \,||\, \underline{n}(x))$ where the argument is that the input is only dependent on the first output and independent on the second. A DCR-graph G that has the same behaviour as the process P is shown in Figure 4.2 (the block event will never be enabled

Fig. 4.2: DCR-graph G simulating the process $P = (\nu n)(\overline{a}\langle n\rangle \,||\, \overline{b}\langle n\rangle \,||\, \underline{n}(x))$



as it is a condition for itself and there to create the disjunctive causality needed). The independence relation for DCR-graphs 1.2.18, in particular the part about them having to not be cause-orthogonal 1.2.17, specify that $n(x)$ is not independent of $a\langle n\rangle$ or $b\langle n\rangle$, meaning no disjunctive causality. To get the disjunctive causality properly identified in our non-interleaving Psi-calculi semantic, we had to split basic events with the events they were linked depending on. A similar thing might be necessary if we choose to have independence for specific markings and not for all possible markings.

Developing a Psi-calculi semantic that takes into account the environmental causality has to take into account that the enabling function for a Psi-calculi instance is possibly different from instance to instance. A proper non-interleaving semantic should give non-interleaving for all possible instances of Psi-calculi, so it must be general enough for different enabling functions. We will further on look into how we might be able to identify environmental dependencies and possibly determine when two transitions are independent.

4.1.1 Main differences between guarded pi-calculus and the Psi-calculi

The main differences between the Psi-calculi and the pi-calculus semantics as given in Chapter 3 is how channels and messages can be shown, and in the way choice is handled through introducing conditions that enable or disable some choices depending on the process's assertions.

In pi-calculus channels and messages can only be single names, in Psi-calculus these can be any nominal term, containing several names, if any names at all. This causes changes in how we can deal with link-causality. In order to handle the link causalities and give causal semantics in pi-calculus, we extended transitions with a D label (see Definition 4.1.4), telling which extruding locations this transition is dependent on. Adding the D label can been seen as splitting of events as for the same non causal transition we could get several causal transitions. We recall the definition bellow.

Definition 4.1.3 (causal semantics): Define the *causal* early semantics as the following transitions:
$(\overline{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ if $(\overline{H}, \underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}', \underline{H}') \vdash P'$ and

1. $D \subseteq \overline{H}$,
2. $(n, l), (n, l') \in D$ implies $l = l'$
3. $\mathsf{dom}(D) = \mathsf{dom}(\overline{H}) \cap no(\alpha)$,

where $\mathsf{dom}(H) = \{n \mid (n, u) \in H\}$, i.e. the set of names recorded in the history, and $no(\alpha)$ is the *non output names* of α, defined by $no(\overline{n}\langle x\rangle m) = \{n\}\backslash\{m\}$, $no(\underline{n}(m)) = \{n, m\}$ and $no(\tau) = \emptyset$.

In Psi-calculus, as it works on terms that possibly contain names, we can still handle the link causality in a similar manner. The main issue is that Definition 3.2.12 assumes at most two names for $no(\alpha)$ where in the general case of Psi-calculi this is a set of arbitrary size. In order to solve this we propose the following change to the definition of $no(\alpha)$ above to

Definition 4.1.4: $no(\alpha)$ is defined as $no(\overline{M}\langle N\rangle) = n(M) \setminus n(N)$,$no(\underline{M}\langle N\rangle) = n(M) \cup n(N)$

and maintain the rest of Definition 3.2.12.

Another major difference between the pi-calculus and the Psi-calculi is how choice branching is handled. There are two differences between the choice in our non-interleaving semantics and the case construct of Psi. The first difference is in going from a guarded syntax to an unguarded one. The second difference is that Psi ad conditions for when a particular branch can be taken in a case construct compared to the choice where all branches are enabled when you get to the choice. In our non-interleaving semantics for pi-calculus given in Chapter 3 we used what is called guarded choice. Guarded choice, or in some versions of pi called guarded sum, is that when there is a choice between $P + Q$, then both P and Q has to be on the form $\varphi.P'$ with φ being either $\underline{a}(x)$ or $\overline{a}\langle n\rangle$. In an unguarded syntax this restriction does not apply, meaning that P and Q can be any valid pi-calculus process. An unguarded syntax allows for nesting of choices as in the process $(((\overline{a}\langle a\rangle + \overline{b}\langle b\rangle) \,||\, \overline{c}\langle c\rangle) + (\overline{d}\langle d\rangle \,||\, (\overline{e}\langle e\rangle + \overline{f}\langle f\rangle)))$.

Going from guarded to unguarded syntax changes how identifying events can be done. We will first look at how we can change our guarded syntax for non-interleaving pi-calculus and get an unguarded syntax. We will then go on to discuss how this may be used to get non-interleaving semantics for Psi-calculi with the addition of conditions to the choices in the Psi-calculi case construct.

The first change we have to do in moving from a guarded to unguarded syntax is to replace the (SUM) rule given in Figure 3.2 with the (TEMPCHOICE) rule given below:

$$\frac{(\overline{H},\underline{H}) \vdash P_i \xrightarrow[u]{\alpha} (\overline{H}',\underline{H}') \vdash P'}{(\overline{H},\underline{H}) \vdash P_0 + P_1 \xrightarrow[u]{\alpha} (\overline{H}',\underline{H}') \vdash P'} \;(\text{TEMPCHOICE})$$

Example 4.1.5 (Choice label): Consider the process $P = (\overline{a}\langle a\rangle.\mathbf{0} \,||\, \overline{b}\langle b\rangle.\mathbf{0}) + (\overline{a}\langle a\rangle.\mathbf{0} \,||\, \overline{c}\langle c\rangle.\mathbf{0})$. Assume now that the pi-calculus semantics we give in Chapter 3 is using the above (TEMPCHOICE) instead of (SUM). In this unguarded syntax we have that the process P has two transitions labelled with the same action $\overline{a}\langle\epsilon\rangle a$ and the same location label $0[\overline{a}\langle a\rangle.\mathbf{0}][\mathbf{0}]$ but reach two different processes, i.e., we have the transitions:

$P \xrightarrow[0[\overline{a}\langle a\rangle.\mathbf{0}][\mathbf{0}]]{\overline{a}\langle a\rangle} \mathbf{0} \,||\, \overline{b}\langle b\rangle.\mathbf{0}$ and $P \xrightarrow[0[\overline{a}\langle a\rangle.\mathbf{0}][\mathbf{0}]]{\overline{a}\langle a\rangle} \mathbf{0} \,||\, \overline{c}\langle c\rangle.\mathbf{0}$.

So, to achieve event determinism as required in Definition 3.4.1(2), we must be able to distinguish between these two transitions. Our suggestion is to add choice labels L, R to the locations in the (CHOICE) rule, distinguishing between taking the left or the right choice. This would in the example above, give us the locations labels $\text{L}0[\overline{a}\langle a\rangle.\mathbf{0}][\mathbf{0}]$ and $\text{R}0[\overline{a}\langle a\rangle.\mathbf{0}][\mathbf{0}]$ for the two transitions.

From the process given in Example 4.1.5 we will, with just changing parts of the (CHOICE) rule, get the transition system in Figure 4.3. As we can see $\overline{b}\langle b\rangle$ generates two different events with different location labels even if it should be the same. One way to handle this is to give a notion of event equivalence, this equivalence would then ignore the choice labels to make the two events equivalent. This would though cause the transitions $P \xrightarrow[\text{L}u]{\alpha} P'$ and $P \xrightarrow[\text{R}u]{\alpha} P''$ to give equivalent events. As this just brings us back to the same problem we showed in Example 4.1.5 this is not a good solution.

Instead we suggest for the (CHOICE) rule to not only add the choice label to the location label but also add the choice label as a marker in the process as shown bellow:

$$\frac{(\overline{H},\underline{H}) \vdash P_{\text{I}} \xrightarrow[u]{\alpha} (\overline{H}',\underline{H}') \vdash P' \qquad \text{I} \in \{\text{L},\text{R}\}}{(\overline{H},\underline{H}) \vdash P_{\text{L}} + P_{\text{R}} \xrightarrow[\text{I}u]{\alpha} (\overline{H}',\underline{H}') \vdash \text{I}P'} \;(\text{CHOICE})$$

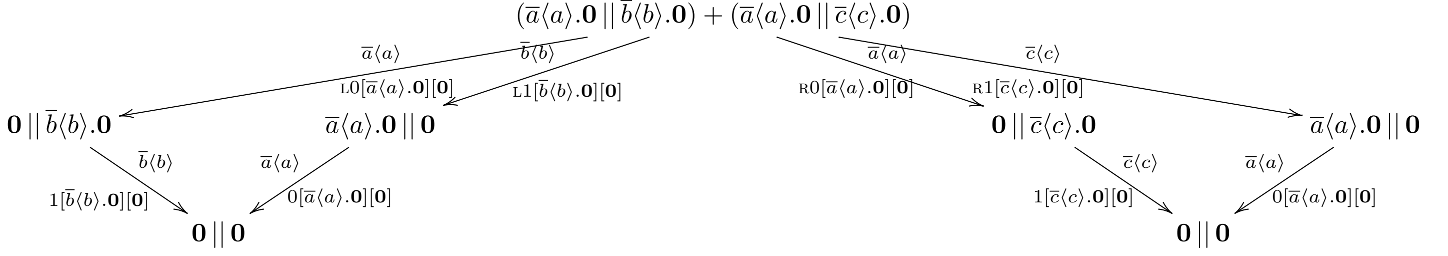


Fig. 4.3: choice labels only from (CHOICE) rule

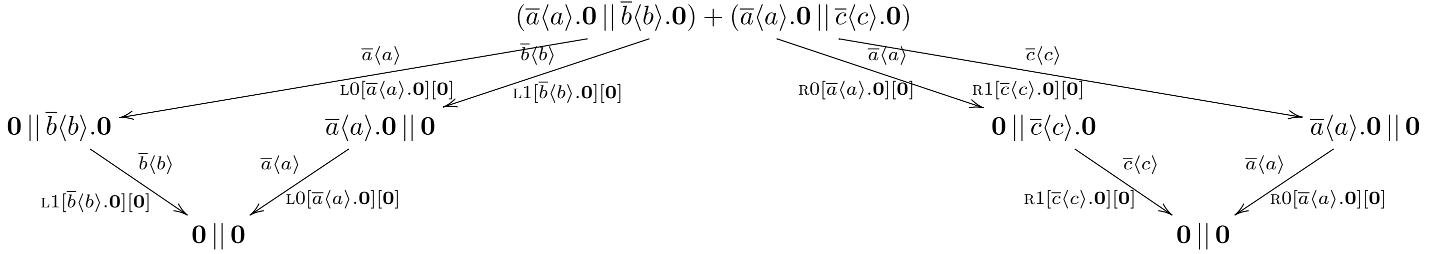


Fig. 4.4: choice labels with (CHOICEHISTORY) and (CHOICE) rules

With just this one rule we would now not add the choice label to transitions from the choice branch other than the initial transition, we would actually not be able to have a single transition from that branch other than the initial transition. To solve these issues we would then add the following rule, which allows transitions of processes with a choice label, and add that choice label to subsequent transitions.

$$\frac{(\overline{H},\underline{H}) \vdash P \xrightarrow[u]{\alpha} (\overline{H}',\underline{H}') \vdash P'}{(\overline{H},\underline{H}) \vdash \textsc{i}P \xrightarrow[\textsc{i}u]{\alpha} (\overline{H}',\underline{H}') \vdash \textsc{i}P'}\ (\textsc{choiceHistory})$$

Adding these changes would give us the transition system in Figure 4.4 from the process in Example 4.1.5.

Moving on to the Psi-calculi case construct, we go from having a binary choice to having an arbitrary amount of cases to choose from. We propose to change the choice labels, instead of being L or R to be the index of the case option used. As this index can be 0 or 1, which already are parts of the location label for parallel branching, just having the index is not enough. Thus we should mark that it is a case label something similar to $(\mathbf{ch}, i)$. This would change Definition 2.2 of prefix locations in Chapter 3 to be

Definition 4.1.6: Let $\mathcal{L} = \{0, 1, (\mathbf{ch}, \varphi_i, i)\}^* \times \mathbf{Proc} \times \mathbf{Proc}$ be the set of *prefix locations* and write $s[P][P']$ for elements in $\mathcal{L}$.

The last part of the main differences is that the case constructs use conditions to determine which case is enabled. While the choice behaviour of pi-calculus can be obtained by having conditions enabled for all assertions, is it also possible to have some or none conditions enabled for some assertions and others enabled for other assertions. While which conditions are enabled tells us what transitions that may happen, they do not directly affect the structural causality other than that each branch of a case construct is in conflict with the other branches. The issue with the conditions show up when we start looking into the environmental-causality. Currently it it is not completely known exactly what information is needed to be remembered in order to generate an environmental independence relation. A suggestion is to remember the condition that are entailed each time the (CASE) rule is used for some transition by having the rule leave behind a case label like I. We will discuss how to discover if two activities are environmentally independent in the next subsection.

4.1.2 Observing environmental-causality in Psi-processes

At this moment we do not know if independence will have to be over all environments called global independence, or if we will work on independence depending on the environments, called local independence. It is easy to see that we can get the global independence relation from a local independence relation by demanding the events to be independent locally for all possible environments. We will come back to this.

The environment of a Psi-calculi process is known as the frame, and is the composition of all the non prefixed assertion appearing in this process. The frame of a process before a transition and after a transition can be different as the reduction may have removed the prefixes of one or more assertions.

Taking the transition $\Psi \rhd P \xrightarrow[u]{\alpha} P'$, we have that the environment before the transition is $\mathcal{F}(P)$ and that after the transition it is $\mathcal{F}(P')$. Knowing that an assertion $(\!|\Psi|\!)$ has no reduction then all assertions making up $\mathcal{F}(P)$ are parts of $\mathcal{F}(P')$. Let us write $\mathcal{F}(P') = \mathcal{F}(P) \otimes \Psi'$ where Ψ' is either $\mathbf{1}$ or the assertions that has had the prefixes removed by the transition.

We can see the process that the (IN) and (OUT) rules reduce to in the location labels. Particularly for some $u = s[P][P']$ the process we get after the (IN) or (OUT) rule is P', and for $u = s\langle 0s_0[P_0][P_0'], 1s_1[P_1][P_1']$ using both rules they are P_0' and P_1'. Based on this knowledge we can define the frame of a transition as follows:

Definition 4.1.7: For any transition $\Psi \rhd P \xrightarrow[u]{\alpha} P'$ we define the frame of the transition $\mathcal{F}((\alpha, u)) = \mathcal{F}(u)$ as follows

1. $\mathcal{F}(s[P][P']) = \mathcal{F}(P')$
2. $\mathcal{F}(s\langle 0s_0[P_0][P_0'], 1s_1[P_1][P_1']\rangle) = \mathcal{F}(P_0') \otimes \mathcal{F}(P_1')$

As independence is on actions/events and not transitions we need to be able to see from the events which assertions that can enable them. First let us recall that an event is on the form $((\alpha, u), D)$. We also recall that in Definition 4.1.6 the case labels contain the condition that was enabled in order to choose that path.

Assuming we have a notion of event enabling $\Psi \vdash e$, we can define the global independence relation as follows:

Definition 4.1.8: $I_g = \{(e, e') \mid \forall \Psi \in \mathbf{A}, (\Psi \vdash e \wedge \Psi \vdash e' \implies \Psi \otimes \mathcal{F}(e) \vdash e' \wedge \Psi \otimes \mathcal{F}(e') \vdash e) \wedge (\Psi \vdash e \wedge \Psi \otimes \mathcal{F}(e) \vdash e' \implies \Psi \vdash e' \wedge \Psi \otimes \mathcal{F}(e') \vdash e)\}$

following the requirements for ATS, and $\mathbf{A}$ is the nominal set of assertions for a Psi-calculi instance.

We can also define local independence as follows:

Definition 4.1.9: $I_l = \{(e, e, \Psi) \mid \Psi \in \mathbf{A}, e, e' \in Ev(\Psi \vdash e \wedge \Psi \vdash e' \implies \Psi \otimes \mathcal{F}(e) \vdash e' \wedge \Psi \otimes \mathcal{F}(e') \vdash e) \wedge (\Psi \vdash e \wedge \Psi \otimes \mathcal{F}(e) \vdash e' \implies \Psi \vdash e' \wedge \Psi \otimes \mathcal{F}(e') \vdash e)\}$

marking in the independence relation what environment the two events e and e' are independent in. From the local independence we should get the global independence relation in the following way:

$I_g = \{(e, e') \mid \forall \Psi \in \mathbf{A} \exists (e, e', \Psi) \in I_l\}$

While using $\mathbf{A}$ in the definition for global independence contains all the assertions that can be in the Psi-calculi instance, it is possible a better idea to use a set containing only reachable assertions. The latter might be better as some instances might have to use some special assertions that never enable transitions in the processes created, but are there to create a valid Psi-calculi instance.

Example 4.1.10 (Entailing events): Assume we have an instantiation of Psi where assertions are sets of names and composition being unions of sets, making $\Psi \otimes \Psi = \Psi$.

Take the processes:

$$P = \Psi \rhd \mathbf{case}\ \varphi\ : \overline{a}\langle a\rangle.\mathbf{0} \mid\mid \mathbf{case}\ \varphi'\ : \overline{b}\langle b\rangle.(\overline{c}\langle c\rangle.(\!|\Psi'|\!) \mid\mid (\!|\Psi'|\!))$$

and

$$P' = \Psi \rhd \mathbf{case}\ \varphi\ : \overline{a}\langle a\rangle.\mathbf{0} \mid\mid \mathbf{case}\ \varphi'\ : (\overline{c}\langle c\rangle.(\!|\Psi'|\!) \mid\mid \overline{b}\langle b\rangle.(\!|\Psi'|\!))$$

where $\Psi \nvdash \varphi \wedge \Psi \vdash \varphi' \wedge \Psi \otimes \Psi' \vdash \varphi$.

It is easy to see that in both P and P' the events coming from $\overline{a}\langle a\rangle$ and $\overline{c}\langle c\rangle$ are structurally independent since they are in parallel branches, and they share no names making them linked independent. In P we have that $\overline{c}\langle c\rangle$

is structurally dependent on $\overline{b}\langle b\rangle$, where in P' this is not so. We can argue for the process P that $\overline{a}\langle a\rangle$ and $\overline{c}\langle c\rangle$ are environmental independent, as the only times one can do either action, the environment is $\Psi \otimes \Psi'$ following the transition for $\overline{b}\langle b\rangle$. In P' we have that $\overline{b}\langle b\rangle$ and $\overline{c}\langle c\rangle$ are independent, as they are structural independent, linked independent and have no way to disable each other environmentally. We cannot claim in P' that $\overline{a}\langle a\rangle$ and $\overline{c}\langle c\rangle$ are environmentally independent as $\overline{c}\langle c\rangle$ may happen before $\overline{b}\langle b\rangle$ causing $\overline{c}\langle c\rangle$ to enable $\overline{a}\langle a\rangle$ by changing the environment.

One problem appearing in Example 4.1.10 is that for both P and P' the suggested semantics makes $\overline{c}\langle c\rangle$ give the exact same event in both processes, specifically: $e = ((\overline{c}\langle c\rangle, 1(\mathbf{ch}, \varphi, 0)0[\overline{c}\langle c\rangle.(\!|\Psi'|\!)][(\!|\Psi'|\!)]), D)$ making it impossible to state if e is enabled or not in Ψ just from looking at the event. In P, e should be enabled in all possible assertions as there is no case construct that it has to pass in order to be enabled, while in P' it may be dependent on the environment asserting φ' to $true$ removing the enabling from all those assertions that this is true for.

The global environmental independence relation we gave in Definition 4.1.8 would in both cases make $\overline{a}\langle a\rangle$ not independent of $\overline{c}\langle c\rangle$ using $\mathbf{A}$. Using only the reachable environments, and not all environments, causes $\overline{a}\langle a\rangle$ and $\overline{c}\langle c\rangle$ not to be independent of each other, due to how the reachable environment Ψ are technically enabling $\overline{c}\langle c\rangle$, but not $\overline{a}\langle a\rangle$.

Another possibility is to only look at the environments that a single event is enabled in. How we are to find the set of environments that enable any single event, is at this moment not clear. It is also not clear how we can formally use the sets of environments, enabling some events e and e', to see if e and e' are environmentally independent. Looking only at an event's possible environments, we should have that the only environment in P, that $\overline{c}\langle c\rangle$ can be executed (and thus be enabled) in, is $\Psi \otimes \Psi'$, which is enabling $\overline{a}\langle a\rangle$, where in P' this is not so. Making $\overline{c}\langle c\rangle$ and $\overline{a}\langle a\rangle$ independent in P but not in P'.

The conclusion from these observations is that the global independence relation given in Definition 4.1.8 is not good, and we need to find one that only looks at the environments that an event may be executed in. This could also possibly help us in knowing in what assertions an event is enabled.

4.1.3 Non-interleaving semantics for Psi-calculi, a suggestion

Our suggestion for providing a non-interleaving semantic for Psi-calculi would be to build upon the work we done for pi-calculus. This means updating the semantics with regards to structural and linked causality to something similar to those given in Figure 4.5. We are confident that this semantic would encompass the structural and linked causalities for a non-interleaving semantic based on those. While the structural causality does not require more, we do have to update the causal semantics to what we proposed in Definitions 3.2.12 and 4.1.4 for the link causality.

The main step forward in the generation of a non-interleaving semantic for Psi-calculi will be to properly generate an independence relation for the environmental causality. We propose that the intersection of the structural independence relation, the linked independence relation and the environmental independence relation will be the true independence relation for Psi-calculi. This can be shown as $eIe' \iff eI_{CCS}\, e' \wedge eI_{\pi}e' \wedge eI_{\Psi}e'$, where I_{CCS} is the structural independence, I_{π} is the linked independence and I_{Ψ} is the environmental independence relations.

4.2 eventPsi and dcrPsi as sanity checks of environmental causality

While obtaining a non-interleaving semantic with the subsequent independence relation for the Psi-calculi has shown itself not to be trivial, we also have to consider how we can be certain that the semantics we reach is correct. While we are confident in the structural and linked independences, we need to have some sanity proofs for the environmental independence.

We have that Event structures and DCR-graphs are both non-interleaving models based on conditional enabling of events. In Chapter 2 we gave instantiations of both these two models into Psi-calculi. A possibility we are working with is to use these instantiations as possible sanity checks for our environmental causality. As both ES and DCR-graphs come with a native independence relation, we have to check that the independence relation we get from the instantiations are the same as from the original model itself. This would demand that we take an ES or DCR-graph, map it into either an eventPsi or dcrPsi process, and then compare the independence relations between the original model and the process it maps into. As we would have to work with processes mapped from original

$$\frac{\Psi \vdash M \dot{\leftrightarrow} K \qquad \overline{H}' = \overline{H} \wedge \underline{H}' = \underline{H} \cup \{(n,\epsilon) \mid n \in fn(\tilde{L})\}}{\Psi \rhd (\overline{H},\underline{H}), \underline{M}(\lambda\tilde{y})N.P \xrightarrow[{[\underline{M}(\lambda\tilde{y})N.P][P[\tilde{y}:=\tilde{L}]]}]{\underline{K}N[\tilde{y}:=\tilde{L}]} (\overline{H}',\underline{H}'), P[\tilde{y} := \tilde{L}]} \text{(IN)}$$

$$\frac{\Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd H, \overline{M}N.P \xrightarrow[{[\overline{M}N.P][P]}]{\overline{K}N} H, P} \text{(OUT)} \qquad \frac{\Psi \rhd H, P \,||\, !P \xrightarrow[u]{\alpha} H', P'}{\Psi \rhd H, !P \xrightarrow[u]{\alpha} H', P'} \text{(REP)}$$

$$\frac{\Psi \rhd H, P_i \xrightarrow[u]{\alpha} H', P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd H, \mathbf{case}\ \tilde{\varphi} : \tilde{P} \xrightarrow[(\mathbf{ch},\varphi_i,i)u]{\alpha} H', (\mathbf{ch},\varphi_i,i)P'} \text{(CASE)}$$

$$\frac{\Psi \rhd H, P \xrightarrow[u]{\alpha} H', P'}{\Psi \rhd H, (\mathbf{ch},\varphi_i,i)P \xrightarrow[(\mathbf{ch},\varphi_i,i)u]{\alpha} H', (\mathbf{ch},\varphi_i,i)P'} \text{(CASEHISTORY)}$$

$$\frac{\begin{array}{c}\Psi \otimes \Psi_Q \rhd (\overline{H}_P,\underline{H}_P), P \xrightarrow[u]{\alpha} (\overline{H}'_P,\underline{H}'_P), P' \qquad bn(\alpha)\#Q \\ \overline{H}'' = \{(n, 0u|_{\{0,1\}^*[P]}) \mid \alpha = \overline{M}N, n \in n(N), (n,[R][R']) \notin \epsilon\overline{H}_\epsilon, (n,l) \in 1\overline{H}_Q, \\ \nexists (n,l) \in 0\underline{H}_P : l \preceq 0u|_{\{0,1\}^*}\}\end{array}}{\begin{array}{c}\Psi \rhd (\epsilon\overline{H}_\epsilon \uplus 0\overline{H}_P \uplus 1\overline{H}_Q, \epsilon\underline{H}_\epsilon \uplus 0\underline{H}_P \uplus 1\underline{H}_Q), P \,||\, Q \xrightarrow[0u]{\alpha} \\ (\epsilon\overline{H}_\epsilon \uplus 0\overline{H}'_P \uplus 1\overline{H}_Q \uplus \overline{H}'', \epsilon\underline{H}_\epsilon \uplus 0\underline{H}'_P \uplus 1\underline{H}_Q), P' \,||\, Q\end{array}} \text{(LPAR)}$$

$$\frac{\begin{array}{c}\Psi \otimes \Psi_Q \rhd H_P, P \xrightarrow[u]{\alpha} H'_P, P' \qquad bn(\alpha)\#Q \\ H'' = \{(n, 1u|_{\{0,1\}^*[P]}) \mid \alpha = \overline{M}N, n \in n(N), (n,[R][R']) \notin \epsilon\overline{H}_\epsilon, (n,l) \in 0H_Q\}\end{array}}{\Psi \rhd \epsilon\overline{H}_\epsilon \uplus 0H_Q \uplus 1H_P, Q \,||\, P \xrightarrow[0u]{\alpha} \epsilon\overline{H}_\epsilon \uplus 0H_Q \uplus 1H'_P \uplus H'', Q \,||\, P'} \text{(RPAR)}$$

$$\frac{\Psi \rhd H, P \xrightarrow[u]{\alpha} H', P' \qquad b\sharp\alpha, \Psi}{\Psi \rhd H, (\nu b)P \xrightarrow[(\nu b)u]{\alpha} H', (\nu b)P'} \text{(SCOPE)} \qquad \frac{\Psi \rhd H, P \xrightarrow[u]{\overline{M}(\nu\tilde{a})N} H', P' \qquad b\sharp\tilde{a}, \Psi, M \qquad b \in n(N)}{\Psi \rhd H, (\nu b)P \xrightarrow[(\dot{\nu} b)u]{\overline{M}(\nu\tilde{a}\cup\{b\})N} H' \uplus \{(b, u|_{\{0,1\}^*[P]})\}, P'} \text{(OPEN)}$$

$$\frac{\begin{array}{c}\Psi_Q \otimes \Psi \rhd H_P, P \xrightarrow[u]{\overline{M}(\nu\tilde{a})N} H'_P, P' \\ \Psi_P \otimes \Psi \rhd H_Q, Q \xrightarrow[v]{\underline{K}N} H'_Q, Q' \qquad \Psi_Q \otimes \Psi_P \otimes \Psi \vdash M \dot{\leftrightarrow} K\end{array}}{\Psi \rhd \epsilon\overline{H}_\epsilon \uplus 0H_P \uplus 1H_Q, P \,||\, Q \xrightarrow[\langle 0u, 1v\rangle]{\tau} \epsilon\overline{H}_\epsilon \uplus 0H_P \uplus 1H_Q, (\nu\tilde{a})(P' \,||\, Q')} \text{(COM)}$$

Fig. 4.5: Semantic rules for Psi-calculi treating the accumulation of extruders information in the histories. (we omitted the symmetric version of rule (COM))

models, and the mapping create syntactic correct processes, we will below discuss how well suited these processes will be for this purpose.

For both these comparisons there is the question of how we look at independence, whether it is global or local, and in the case of global, how we define the set of all environments for a single event, or process.

4.2.1 Event structure independence in Psi-calculi

The environment of an event structure can be considered as the configuration of an event structure. Knowing an events conditions and conflicts, we can for any configuration say if it has happened or can happen. As we see in Remark 1.2.9 we have that the configurations determine the conditions and conflicts. That the configurations determine conditions and conflicts makes it impossible for there to be a configuration $\{a, b\}$ if $a\sharp b$, making configurations the set of reachable environments. For the eventPsiprocess instance this does not make much of a difference as **A** are the configurations, not any set of events in the ES.

We recall that in the syntactic restrictions for the eventPsiinstance, each event is given a sub-process placed in parallel with the other events sub-processes. Each sub-process for some event e consists of a case construct, with the conditions being the events e depends on and those e is in conflict with. The event transition is a single output guarded by the case construct, using the events name as channel, reducing to an assertion containing the event name.

For structural causality the independence relation would be all possible pairs of events in the event structure. This comes from all events being parallel to each other and thus no prefixing giving rise to any structural causal relations.

For linked dependencies we also have that its independence relation will be all possible pairs of events. This comes from there being only output actions possible, and no restricted names.

Both these things gives us that for the full independence we will have to look into the environmental causality and that due to both the structural and linked independences being the largest possible we have that the full independence is the same as the environmental independence.

4.2.2 DCR-graph independence in Psi-calculi

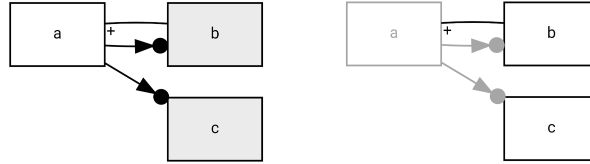
The environment of a DCR-graph can be considered to be the marking. The set of markings are defined as all possible triples where each element is a set of events. This is the same as all possible environments where for a single DCR-graph only a subset of the markings are reachable from a set start marking. The independence relation for DCR-graphs are defined over relations between events. This means that the independence are over all possible markings not only over reachable markings. It is possible to create a DCR-graph where we have an event that is effect- or cause-orthogonal to another event, but the graph never reaches a marking where this has any impact on enabled or demanded events.

An example of this is the left DCR-graph in Figure 4.6 where it is easy to see that b and c are not cause-orthogonal, as b includes a which is a condition for c. In the initial marking shown then b does not affect the inclusion of a in any way, and further, the only event possible at the start is a due to it being a condition for both b and c. Once a has happened at least once, the condition is fulfilled and both c and b can happen in any order, and should be independent for the marking reachable in this graph. A marking where they are not independent is the one where a is initially excluded shown as the right DCR-graph in Figure 4.6. In this graph c can happen at will until b happens, in which case both b and c has to wait on a in order to be able to happen again.

For dcrPsi, we recall that the syntactic restrictions are on the form of one output activity over channel m, that reduces to the empty process. For each event in the DCR-graph we have one replicating process consisting of a case-guarded input over channel m reducing to an output of same structure as the single output we have, and an assertion that is in parallel with the output. There is one or more top level assertions in parallel to the output and inputs. Everything is placed under a restriction on the channel name m which also is unique compared to all the events.

When looking at the structural causality, we can see from the syntactic restriction of dcrPsithat it has only one channel m and that it is bound to the process. This means that it is not possible to have a transition leaving the scope of the process, due to the $b\sharp\alpha$ and $b\sharp M, \tilde{a}$ restrictions in the (SCOPE) and (OPEN) rules. As we cannot have

Fig. 4.6: Left DCR-graph the non-orthogonal events b and c are arguably independent. In the right they are clearly not independent as b will disable c by including a



any transitions out of the scope, all transitions must be τ transitions using the (COM) rule. By Lemma 4.19 in Chapter 2 we have that there is always only one output. By only being able to have τ transitions and the input reducing to a new output we can say for some process P that if we have two event transitions (see Definition 4.23 in Chapter 2) with labels a and b enabled in P then they must share the same output and thus be structural dependent. If we have a trace ab from P, then the output location of b is structural dependent on a. This means that for any two transitions that are structural independent, they can not be enabled in the same process, nor can they be directly after each other in a trace language. This in turn makes I_{CCS} not interesting when comparing with the independence relation of DCR-graphs, due to events that are independent in DCR are causual structural dependent in our instantiation.

For the linked-causality we have that there is only one bound name, and that name is not possible to be extruded. This means that as there is no events being enabled by an extruder there is no linked causality, making $I_\pi = \{(e, e') \mid e, e' \in E, e \neq e'\}$ for the set of events E in the DCR-graph we built the process from.

For the environment-causality there are some observations we have to see first. In DCR-graphs, what an event is does not care about the event just before when it comes to its label. In our Psi-calculi instantiation this is not completely true for events as defined in our non-interleaving semantic, while it is true for event transitions that get relabelled according to Definition 4.23 in Chapter 2. In non-interleaving Psi, we would, for our instantiation and having the events a, b, c in DCR-graph, get different events for c depending on if the trace leading to it is abc or bac, due to the communications having different locations on the output part. If we are to use the instance of DCR-graphs in Psi-calculi as a possible sanity check for our environmental causality, we have first to find a way to redefine the events to make all events for c in this small example be equivalent to each-other while maintaining all the information about the environmental causality as we need.

While this could be done declaring that all transitions that give the same event-transition label to be equivalent transitions, and ignore the location of the output parts, it is not enough. Initially we would get that any two equivalent transitions would have the same case guards, just in possible different amounts of guards, making it that any assertion enabling one would automatically enable the other. The problem comes with the frame of the transition. Due to how the upgrade of assertions are handled in our instantiation, and the markings in DCR-graphs, we have that the frame of the events is completely dependent on the frame of the entire process before the event. As we suggest to use the frame of an event as parts of the information drawn from events to determine if it is environmental independent of another event, this causes us to not be able to get that information properly. For our instantiation of DCR-graphs we used generations making a completely new generation of the environment for each transition, while normally frame of an event is the update that the event does to the environment instead of the entire new environment. The only way to be able to use the instantiation of DCR-graphs as a sanity check would be to only look at how the environment changes with an event, instead of the frame of the event.

This gives us that while using the event structures as a sanity check would be quite easy, the same cannot be said about DCR-graphs. For DCR-graphs we have to change the events we get from the transitions, define equivalence classes on events and find a different way of defining the frame of events.

Getting the same independence relation would also demand that we look at all possible assertions, and not just reachable assertions. Assuming the comparison works out, this can make for giving an independence relation to DCR-graphs using only reachable markings instead of all markings, by changing from all assertions to reachable assertions.

4.3 Conclusion and further work

In this thesis we have presented encodings of the declarative event-based models of concurrency of finite prime event structures and finite DCR-graphs into corresponding instances of Psi-calculi. We gave syntactic restriction to both instances and a mapping from the original model into a Psi-process respecting the syntactic restrictions. We showed that any step in the model corresponds to a transition in the process the model was mapped to. For both models we made use of the expressive logic of Psi-calculi, while we for the dcrPsi instance had to use communication due to the non-monotonicity of the markings enabling transitions.

For the DCR instance we had to observe the behaviour by a somewhat intentionally constructed event-labelled transition system. This is not completely satisfactory and an improvement could be to consider a more compositional definition of event structures and DCR graphs as considered in [DHS15a]. It would also be interesting to look into adding responses to Psi-calculi as introduced in DCR graphs, allowing to represent liveness/progress properties, continuing along the lines of the work in [CHPW12] for Transition Systems with Responses.

We have also provided the first stable non-interleaving operational semantics for the full early pi-calculus. Where others have given non-interleaving semantics for pi-calculus before they have all been for the late pi-calculus, and most have not considered the choice like in [Cri15] or developed denotational semantics and not operational [CVY12]. We have seen that the addition of choice is not directly trivial, this becomes more prominent when one looks at unguarded choice compared to guarded choice. We also see that going from late to early semantics provide its own problems, specifically we had to include a history of names received, as these are decided in the (INPUT) rule and not delayed to the (COM) rule.

One thing working with developing the non-interleaving semantics showed was that we had to go away from the classic notion of having extruder's in the transition labels. If we tried to keep them in the labels we could, for a single event, get several different transition label's due to scope extrusion or scope extensions. The simplest example being $(\nu n)(\overline{a}\langle n\rangle \,||\, \overline{b}\langle n\rangle)$ where the first transition would get the extruder and the second would not. Instead of having them in the transition label we moved the extruder's to the extruder history and showed that by checking the histories before and after a transition would give us the extruder just as if we had them in the label.

Moving on from this we would like to make a comparison between our semantics and the denotational event structure semantics for the late pi-calculus done by Varaca, Crafa and Yoshida [CVY12]. We have already provided insight into the work done to extend the non-interleaving pi-calculus semantics to the meta-language of Psi-calculi. The first main part of providing non-interleaving semantics for Psi-calculi will be to properly understand what environmental causality is, and how it behaves in terms of independence. This would not only affect how one looks at non-interleaving for Psi-calculi but any model of concurrency where the environment or state of the process is directly affecting what may happen, like the markings for DCR-graphs. The stage after that in developing non-interleaving semantics for Psi-calculi is to find a proper way to see what a transition does to the environment, and how one transition might enable or disable another.

Steps towards studying adaptable, distributed and mobile computational artefacts will be to consider cases of workflows identified in field studies within the CompArt project and the notion of run-time refinement and adaptation supported by DCR graphs as presented in [DHS15a]. By embedding DCR graphs in the richer framework of Psi-calculi we anticipate being able to experiment with richer process models, e.g. representing locations, mobility and resources used by actors in workflows.

BIBLIOGRAPHY

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL*, pages 104–115. ACM, 2001.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.

[AGNV07] Jesús Aranda, Cinzia Di Giusto, Mogens Nielsen, and Frank D. Valencia. CCS with Replication in the Chomsky Hierarchy: The Expressive Power of Divergence. In Zhong Shao, editor, *5th Asian Symposium on Programming Languages and Systems (APLAS)*, volume 4807 of *LNCS*, pages 383–398. Springer, 2007.

[AGPV07] Jesús Aranda, Cinzia Di Giusto, Catuscia Palamidessi, and Frank D. Valencia. On Recursion, Replication and Scope Mechanisms in Process Calculi. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects (FMCO'06)*, volume 4709 of *LNCS*, pages 185–206. Springer, 2007.

[BCH+14] Mario Bravetti, Marco Carbone, Thomas T. Hildebrandt, Ivan Lanese, Jacopo Mauro, Jorge A. Pérez, and Gianluigi Zavattaro. Towards global and local types for adaptation. In Steve Counsell and Manuel Núñez, editors, *Software Engineering and Formal Methods (SEFM)*, volume 8368 of *LNCS*, pages 3–14. Springer, 2014.

[BCHK94] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. A theory of processes with localities. *Formal Asp. Comput.*, 6(2):165–200, 1994.

[Bed87] Marek Antoni Bednarczyk. *Categories of Asynchronous Systems.* PhD thesis, Sussex, UK, UK, 1987. AAIDX83002.

[Bed88] Marek A. Bednarczyk. *Categories of asynchronous systems.* PhD thesis, Univ. Sussex, 1988.

[BG95] Nadia Busi and Roberto Gorrieri. A Petri Net Semantics for pi-Calculus. In *CONCUR*, volume 962 of *LNCS*, pages 145–159. Springer, 1995.

[BGP+14] Johannes Borgström, Ramunas Gutkovas, Joachim Parrow, Björn Victor, and Johannes Åman Pohjola. A sorted semantic framework for applied process calculi (extended abstract). In Martín Abadi and Alberto Lluch-Lafuente, editors, *8th International Symposium on Trustworthy Global Computing (TGC)*, volume 8358 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2014.

[BGP+15] Johannes Borgström, Ramunas Gutkovas, Joachim Parrow, Björn Victor, and Johannes Åman Pohjola. A sorted semantic framework for applied process calculi. *Logical Methods in Computer Science (LMCS)*, abs/1510.01044, 2015. (submitted).

[BGPZ12] Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Adaptable processes. *Logical Methods in Computer Science*, 8(4), 2012.

[BHJ+11] Johannes Borgström, Shuqin Huang, Magnus Johansson, Palle Raabjerg, Björn Victor, Johannes Åman Pohjola, and Joachim Parrow. Broadcast psi-calculi with an application to wireless protocols. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *9th International Conference on Software Engineering and Formal Methods (SEFM)*, volume 7041 of *LNCS*, pages 74–89. Springer, 2011.

[BJPV11] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1), 2011.

[BM07] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *ESOP'07*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BS95] Michele Boreale and Davide Sangiorgi. A fully abstract semantics for causality in the pi-calculus. In *STACS*, pages 243–254, 1995.

[Cas01] Ilaria Castellani. Process algebras with localities. In *Handbook of Process Algebra*, chapter 15, pages 945–1045. Elsevier, 2001.

[CGMP99] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Int. Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[CHPW12] Marco Carbone, Thomas T. Hildebrandt, Gian Perrone, and Andrzej Wasowski. Refinement for transition systems with responses. In *FIT*, volume 87 of *EPTCS*, pages 48–55, 2012.

[CKV13] Ioana Cristescu, Jean Krivine, and Daniele Varacca. A compositional semantics for the reversible pi-calculus. In *ACM/IEEE Symposium on Logic in Computer Science, LICS*, pages 388–397. IEEE Computer Society, 2013.

[CM03] Marco Carbone and Sergio Maffeis. On the Expressive Power of Polyadic Synchronisation in pi-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.

[Cri15] Ioana Cristescu. *Operational and denotational semantics for the reversible π- calculus.* PhD thesis, Université Paris Diderot - Paris 7 - Sorbonne Paris Cité, 2015.

[CS00] Gian Luca Cattani and Peter Sewell. Models for Name-Passing Processes: Interleaving and Causal. In *LICS*, pages 322–333. IEEE Computer Society, 2000.

[CVY12] Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. Event structure semantics of parallel extrusion in the pi-calculus. In *FOSSACS*, volume 7213 of *LNCS*, pages 225–239. Springer, 2012.

[DDM88] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. Partial orderings descriptions and observations of nondeterministic concurrent processes. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, REX Proceedings*, volume 354 of *LNCS*, pages 438–466. Springer, 1988.

[DFM+05] Rocco De Nicola, Gian Luigi Ferrari, Ugo Montanari, Rosario Pugliese, and Emilio Tuosto. A Process Calculus for QoS-Aware Applications. In *COORDINATION*, volume 3454 of *LNCS*, pages 33–48. Springer, 2005.

[DGK00] Manfred Droste, Paul Gastin, and Dietrich Kuske. Asynchronous cellular automata for pomsets. *Theor. Comput. Sci.*, 247(1-2):1–38, 2000.

[DHS15a] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In Nikolaj Bjørner and Frank de Boer, editors, *20th International Symposium on Formal Methods (FM)*, volume 9109 of *LNCS*, pages 143–161. Springer, 2015.

[DHS15b] Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. Concurrency and asynchrony in declarative workflows. In *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, pages 72–89, 2015.

[DHSM14] Søren Debois, Thomas Hildebrandt, Tijs Slaats, and Morten Marquard. A Case for Declarative Process Modelling: Agile Development of a Grant Application System. In Georg Grossmann, Sylvain Hallé, Dimka Karastoyanova, Manfred Reichert, and Stefanie Rinderle-Ma, editors, *18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOC Workshops)*, pages 126–133. IEEE, 2014.

[DP99] Pierpaolo Degano and Corrado Priami. Non-interleaving semantics for mobile processes. *Theor. Comput. Sci.*, 216(1-2):237–270, 1999.

[GH05] Jens Chr. Godskesen and Thomas T. Hildebrandt. Extending Howe's Method to Early Bisimulations for Typed Mobile Embedded Resources with Local Names. In Ramaswamy Ramanujam and Sandeep Sen, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 3821 of *LNCS*, pages 140–151. Springer, 2005.

[GM92] Joseph A. Goguen and José Meseguer. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[God96a] Patrice Godefroid. On the costs and benefits of using partial-order methods for the verification of concurrent systems. In *Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, July 24-26, 1996*, pages 289–304, 1996.

[God96b] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[GP02] Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

[Gup94] Vincent Gupta. *Chu Spaces: A Model of Concurrency.* PhD thesis, Stanford University, 1994.

[GW91] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings*, pages 332–342, 1991.

[Hen07] Matthew Hennessy. *A Distributed Pi-Calculus.* Cambridge Univ. Press, 2007.

[Hil99] Thomas Troels Hildebrandt. *Categorical Models for Concurrency: Independence, Fairness and Dataflow.* PhD thesis, University of Aarhus, Denmark, 1999.

[HJN17] Thomas Troels Hildebrandt, Christian Johansen, and Håkon Normann. A stable non-interleaving early operational semantics for the pi-calculus. In *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, pages 51–63, 2017.

[HM10] Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *PLACES*, volume 69 of *EPTCS*, pages 59–73, 2010.

[HMS12] Thomas T. Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Nested dynamic condition response graphs. In *FSEN*, volume 7141 of *LNCS*, pages 343–350. Springer, 2012.

[Hüt11] Hans Hüttel. Typed psi-calculi. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2011.

[Hüt14] Hans Hüttel. Types for resources in ψ-calculi. In Martín Abadi and Alberto Lluch-Lafuente, editors, *8th International Symposium on Trustworthy Global Computing (TGC)*, volume 8358 of *LNCS*, pages 83–102. Springer, 2014.

[JJ95] Lalita Jategaonkar Jagadeesan and Radha Jagadeesan. Causality and True Concurrency: A Data-flow Analysis of the Pi-Calculus. In *AMAST*, volume 936 of *LNCS*, pages 277–291. Springer, 1995.

[Joh15] Christian Johansen. ST-structures. *The Journal of Logical and Algebraic Methods in Programming (JLAMP)*, December 2015.

[LMS16] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversibility in the higher-order pi-calculus. *Theoretical Computer Science*, 625:25–84, 2016.

[MHS13] Raghava Rao Mukkamala, Thomas Hildebrandt, and Tijs Slaats. Towards trustworthy adaptive case management with dynamic condition response graphs. In *17th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pages 127–136. IEEE, 2013.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983.

[Mil92] Robin Milner. Functions as Processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MN92] Madhavan Mukund and Mogens Nielsen. CCS, Location and Asynchronous Transition Systems. In *FSTTCS*, volume 652 of *LNCS*, pages 328–341. Springer, 1992.

[MP95] Ugo Montanari and Marco Pistore. Concurrent semantics for the pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 1:411–429, 1995.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I-II. *Information and Computation*, 100(1):1–77, 1992.

[MPW93] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theor. Comput. Sci.*, 114(1):149–171, 1993.

[NJH16a] Håkon Normann, Christian Johansen, and Thomas Hildebrandt. Non-interleaving operational semantics for the pi-calculus (long version). Technical Report 453, Dept. Info., University of Oslo, 2016. `http://heim.ifi.uio.no/~cristi/papers/TR453.pdf`.

[NJH16b] Håkon Normann, Christian Johansen, and Thomas T. Hildebrandt. Declarative event based models of concurrency and refinement in psi-calculi. *J. Log. Algebr. Meth. Program.*, 85(3):368–398, 2016.

[NPH14] Håkon Normann, Cristian Prisacariu, and Thomas Hildebrandt. Concurrency Models with Causality and Events as Psi-calculi. In Ivan Lanese, Alberto Lluch-Lafuente, Ana Sokolova, and Hugo Torres Vieira, editors, *7th Interaction and Concurrency Experience (ICE 2014)*, volume 166 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 4–20. Open Publishing Association, 2014.

[NPW79] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains. In *Semantics of Concurrent Computation*, volume 70 of *LNCS*, pages 266–284. Springer, 1979.

[Pel93] Doron A. Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, pages 409–423, 1993.

[PGG+15] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Developing correct, distributed, adaptive software. *Science of Computer Programming*, 97, Part 1:41 – 46, 2015. Special Issue on New Ideas and Emerging Results in Understanding Software.

[Pit13] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge Univ. Press, 2013.

[Pra91] Vaughan R. Pratt. Modeling concurrency with geometry. In *POPL'91*, pages 311–322, 1991.

[Pra95] Vaughan R. Pratt. Chu spaces and their interpretation as concurrent objects. In *Computer Science Today: Recent Trends and Develop.*, volume 1000 of *LNCS*, pages 392–405. Springer, 1995.

[Pra00] Vaughan R. Pratt. Higher dimensional automata revisited. *Math. Struct. Comput. Sci.*, 10(4):525–548, 2000.

[Qua99] Paola Quaglia. The pi-calculus: Notes on labelled semantics. *Bull. EATCS*, 68:104–114, 1999.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes (extended abstract). In David B. MacQueen and Luca Cardelli, editors, *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 378–390. ACM, 1998.

[RHT08] M.R. Rao, T. Hildebrandt, and J.B. Tøth. The resultmaker online consultant: From declarative workflow management in practice to LTL. In *12th Enterprise Distributed Object Computing Conference Workshops*, pages 135 – 142. IEEE, 2008.

[RW12] Manfred Reichert and Barbara Weber. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies.* Springer, 2012.

[San96] Davide Sangiorgi. Locality and interleaving semantics in calculi for mobile processes. *Theor. Comput. Sci.*, 155(1):39–83, 1996.

[Shi85a] M. W. Shields. Concurrent machines. *Comput. J.*, 28(5):449–465, 1985.

[Shi85b] M. W. Shields. Concurrent machines. *Computer Journal*, 28(5):449–465, 1985.

[Sla15] Tijs Slaats. *Flexible Process Notations for Cross-organizational Case Management Systems.* PhD thesis, IT University of Copenhagen, January 2015.

[SMHM13] Tijs Slaats, Raghava Rao Mukkamala, Thomas T. Hildebrandt, and Morten Marquard. Exformatics declarative case management workflows as dcr graphs. In *Business Process Management*, volume 8094 of *LNCS*, pages 339–354. Springer, 2013.

[SNW96] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models for concurrency: Towards a classification. *Theor. Comput. Sci.*, 170(1-2):297–348, 1996.

[SW01] Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes.* Cambridge Univ. Press, 2001.

[UPG04] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.

[UPY14] Irek Ulidowski, Iain Phillips, and Shoji Yuen. Concurrency and reversibility. In *Reversible Computation: 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings*, volume 8507 of *LNCS*, pages 1–14. Springer, 2014.

[Val89] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, pages 491–515, 1989.

[Val90] Antti Valmari. A stubborn attack on state explosion. In *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, pages 156–165, 1990.

[vG06] Rob J. van Glabbeek. On the Expressiveness of Higher Dimensional Automata. *Theor. Comput. Sci.*, 356(3):265–290, 2006.

[vGG01] Rob van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001.

[vGP09] Rob van Glabbeek and Gordon Plotkin. Configuration structures, event structures and Petri nets. *Theor. Comput. Sci.*, 410(41):4111–4159, 2009.

[vGV97] Rob J. van Glabbeek and Frits W. Vaandrager. The Difference between Splitting in n and n+1. *Information and Computation*, 136(2):109–142, 1997.

[Win80] Glynn Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[Win82] Glynn Winskel. Event Structure Semantics for CCS and Related Languages. In *ICALP*, volume 140 of *LNCS*, pages 561–576. Springer, 1982.

[Win87] Glynn Winskel. Event Structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets: Central Models and Their Properties, Part II*, volume 255 of *LNCS*, pages 325–392. Springer, 1987.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of ACM*, 14(4):221–227, 1971.

[WN95] Glynn Winskel and Mogens Nielsen. Models for concurrency. In S. Abramski, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 1–148. Oxford, 1995.