UiO : Department of Informatics
University of Oslo

Event based True Concurrency Models and Psi-calculi

Håkon Normann

Master's Thesis Spring 2014

Event based True Concurrency Models and Psi-calculi

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science (MSc)
to the Faculty of Mathematics and Natural Sciences,
at the University of Oslo.

Håkon Normann
Precise Modeling and Analysis Group (PMA)
Department of Informatics, University of Oslo

Supervisors:
Dr. Cristian Prisacariu, *University of Oslo*
Assoc. Prof. Thomas Hildebrandt, *IT University of Copenhagen*
Prof. Olaf Owe, *University of Oslo*

Thesis committee:
Dr. Paolo Torrini, *Swansea University*
Prof. Herman Ruge Jervell, *University of Oslo*

May 2, 2014

Abstract

Psi-calculi are a parametric framework for nominal calculi, where standard calculi are found as instances, like the pi-calculus, or the cryptographic spi-calculus and applied-pi. Psi-calculi have an interleaving operational semantics, with a strong foundation on the theory of nominal sets and process algebras. Much of the expressive power of psi-calculi comes from their logical part, i.e., assertions, conditions, and entailment, which are left quite open thus accommodating a wide range of logics. We are interested in how this expressiveness can deal with event-based models of concurrency. We thus take the popular prime event structures model and give an encoding into an instance of psi-calculi. We also take the recent and expressive model of Dynamic Condition Response Graphs (in which event structures are strictly included) and give an encoding into another corresponding instance of psi-calculi. The encodings that we achieve look rather natural and intuitive. Additional results about these encodings give us more confidence in their correctness.

The work in this thesis has also been published or/and presented at the following international workshops:
- "Event Structures as Psi-calculi" at NWPT, November 2013
- "Concurrency Models with Causality and Events as Psi-calculi" at ICE, June 2014
- "True Concurrency Semantics for Psi-calculi" at MeMo, June 2014

Acknowledgements!

Most importantly I would like to warmly thank Cristian Prisacariu for all the discussions we have had during the master project, and for encouraging me to submit my findings to workshops. I would also like to thank Thomas Hildebrandt who was my supervisor during my stay at IT University of Copenhagen as an Erasmus student.

Thanks to all my family and friends who stayed with me during this project.

Contents

Chapter 1

Introduction

1.1 Introduction

We look into true concurrency models as they, in contrasts to interleaving models [Mil80], allow for multiple actions to happen simultaneously. In the real world things do happen at the same time, and then true concurrency seams more adaptable to be used as a model for a wider range of situations. In the computer world interleaving is good for modelling single core computers. But nowadays we can see a clear trend towards multi-core platforms, hence one easily arrives in situations where processes happen at the same time and thus the interleaving model is less accurate.

Psi-calculi [BJPV11] is a recent framework where various existing calculi can be found as instances. In particular, the spi- and applied-pi calculi [AG99, AF01] are two instances of interest for security. Psi-calculi can also accommodate probabilistic models, by going through CC-pi [BM07, DFM$^+$05] which has already been treated as a corresponding psi-calculus instance. The theory of psi-calculi is based on nominal data structures [Pit13]. Psi-calculi can be seen as a generalization of pi-calculus with two main features:

(i) data structures (i.e., general, possibly open, terms) in place of communication channels and also in place of the communicated data; and

(ii) a rather open logic for capturing dependencies (i.e., through conditions and entailment) on the environment (i.e., assertions) of the processes.

The semantics of psi-calculi is given through structural operational rules and adopts an interleaving approach to concurrency, in the usual style of process algebras. On the other hand, event-based models of concurrency take a non-interleaving view. Many times these form domains and are used to give denotational semantics, as e.g., done by Winskel in [Win82, WN95]. Many times non-interleaving models of concurrency can actually distinguish between interleaving and, so called, "true" concurrency, as is the case with higher dimensional automata [Pra91, Pra00, vG06], configuration structures [vGP09], or Chu spaces [Gup94, Pra95]. The recent Dynamic Condition Response graphs (abbreviated DCR-graphs or DCRs) [HM10a] is a model of concurrency with high expressive power which strictly extends event structures by refining the notions of dependent and conflicting events, and including the notion of response. Due to their graphical nature, DCRs have been successfully used in industry to model business processes [SMHM13].

Research goals

In Chapter 3 a first goal we are interested in is how psi-calculi could accommodate the event structures model of concurrency [NPW79, Win86], with a final goal of capturing the DCR-graphs model [HM10a], that we explore in Chapter 4. Event names in event-based models of concurrency are unique, and can thus be thought of nominals, whereas the execution of an event can be seen as a communication of some sort. The dependencies between events that an event structure defines can be captured with assertions on the nominal data structures, whereas the notion of computation is captured through reduction steps between psi-processes. To be confident on the encodings, we like to see a correlation between the notions of concurrency from the two encoded models and the interleaving diamonds from the psi-calculus behaviour.

These are the basic ideas we follow in this work to give encodings of event structures and DCRs into corresponding instances of psi-calculus which we call respectively eventPsi and dcrPsi. After a few results meant to better explain the correlation between the encoding and the event structure model, we give a result that shows that the concurrency embodied by the event structure is captured in the encoding psi-process through the standard interleaving diamond. For the event structures encoding we also give a result that identifies the syntactic shape of those psi-processes which correspond exactly to event structures. Another feature of true concurrency models is that they are well behaved wrt. action refinement [vGG01]. For this we give a result showing that action refinement is preserved by our translation under a properly defined refining function on psi-processes, which we define similarly to the refinement function on the event structures.

A last goal is to define non-interleaving semantics for psi-calculi. This is still ongoing work which is presented here in Chapter 5.

The semantics of psi-calculi is given through structural operational [Plo81] rules and adopts an interleaving approach to concurrency, in the usual style of process algebras. On the other hand, event-based models of concurrency take a non-interleaving view. Many times these form domains and are used to give denotational semantics, as e.g., done by Winskel in [Win82, WN95]. Many times non-interleaving models of concurrency can actually distinguish between interleaving and, so called, “true concurrency”, as is the case with higher dimensional automata [Pra91, Pra00, vG06], configuration structures [vGP09], or Chu spaces [Gup94, Pra95].

We are interested in non-interleaving semantics for psi-calculi, and Chapter 5 reports on preliminary results in this direction. In particular, we are interested in a true concurrency semantics that is more operational than denotational. In other words, we would like the concurrency model that is obtained from the semantics of psi-calculus to be less like event structures or configurations structures, and more like higher dimensional automata or asynchronous transition systems [Bed88, Shi85].

1.2 Motivations and design decisions

The work in this thesis is done in order to investigate the expressiveness of the psi-calculi [BJPV11] meta language in connection with the expressiveness of true concurrency models. Psi-calculi differentiates itself from other process calculi through the fact that it is not a single calculus in itself, but rather a framework that will give a correct process calculus when a correct instantiation is done. Along with this is the fact that it has been formally proved using the Nominal Isabel theorem proofer [Ben12].

The starting idea for this thesis was to define a psi-calculi instance that would accommodate the true concurrency model DCR-graphs [HM10b]. As a preliminary step towards this goal we chose to work on a more basic true concurrency model known as prime event structures, for which we defined a psi-calculi instantiation that we called eventPsi.

When we first started developing the eventPsi instance, the main focus of the master project was to look into distribution of DCR-graphs, and how this could be used to make a true-concurrent programming language. The work of instantiating DCR-graphs into psi-calculi was started during the stay of this author at ITU Copenhagen as an Erasmus student, where he was taught about psi-calculi by the Uppsala authors [BJPV11] who were visiting ITU at that time. In addition to just instantiating DCR-graphs into psi-calculi, there was also the hope that we could use this instantiation as a help in distributing DCR-graphs. This hope affected the preliminary approaches to our instantiation, but was abandoned as the initial attempts that might given a distribution help were shown not to promising. The work on instantiating both event structures and then DCR-graphs took more time than initially expected, while also proving to be more interesting than expected. As it was becoming more interesting, and time was of a factor, the focus for the thesis moved away from the distribution aspect and more into exploring psi-calculi in more depth. An idea that came up early after the focus had shifted was to look into how to adapt the operational rules of psi-calculi to give it a non-interleaving semantic without having to change the proven work done for psi-calculi.

When we started to work on instantiating event structure into psi-calculi, the original thought was to use an extension to psi-calculi known as broadcast psi [BHJ$^+$11]. This adds two new predicates to psi-calculi, that have to be instantiated (refer to Section 2.1 for the definitions of psi-calculi). These two predicates are:

$\overset{\bullet}{\prec}: T \times T \to C$ Output Connectivity

$\overset{\bullet}{\succ}: T \times T \to C$ Input Connectivity

The first predicate allows a single output over some channel a to be sending over a broadcast channel b, i.e. $a \overset{\bullet}{\prec} b$. In a similar way does the second predicate allows all enabled processes with an input on some channel c to be receiving at the same time a message outputted on b, i.e. $b \overset{\bullet}{\succ} c$. This makes it possible for a single output process to synchronize with every input process at the same time with the same message.

The idea was primarily to use this and have each event just send on a global broadcast channel a message signifying that it had happened. In this way each event would have a replicating input process, and a single output process. The input process would constantly be listening for messages on the broadcast channel about events happening, and then update a local assertion that the event it was input for could use to determine if it was enabled. This idea was thought of as a way to simulate the message passing between different computers or other entities that events might be distributed over, where broadcast messages would be messages sent to everyone on the network.

A different but similar approach that was considered at the same time, was instead of having a single broadcast channel, there would be one for each event. In this approach events would only listen to the events they were depending on to know when they were enabled or not.

Both ideas had several problems. The main one was how to make a local assertion that would only affect its sub-process and not all the other sub-processes. This turned out not to be possible. A second problem was that this approach could possibly become quite more complex than necessary.

The idea of using assertions for deciding when an event/process should be able to happen was there from the beginning, but has changed to be one global assertion. Because of this we went away from broadcast psi as its extra predicates were no longer needed to make the assertion affect the sub-processes the way we wanted. We still had a communication between two processes as the sign of an event happening, but this would leave us with a transition system with only τ transitions, a notion that was not preferable. Moreover as an output process does not need to synchronize with any other process in order to have a reduction step, we could remove the communication aspect altogether. This left us with a natural and intuitive solution: For each event we would have an output process guarded by a condition, who on a transition would reduce into an assertion, with the events name as both transition label and as the assertion left behind.

This gave us the nice simple instantiation we present here in Chapter 3.

After we had made eventPsi, we turned our attention to creating dcrPsi as an instantiation of DCR-graphs into psi-calculi. As DCR-graphs is a conservative generalization of event structures, the idea was to use the same methods as we used in the instantiation of eventPsi as much as possible. There were some problems that came from the fact that DCR-graphs are more complex than event structures. This is visible primarily in the *marking* which is the analogue of the configuration in event structures, i.e., the state of the system. In eventPsi we found a nice way to make the assertions be the same as the configurations. But for DCR-graphs problems came from the requirements to how the composition of assertions must be instantiated to be valid in psi-calculi, and how the marking of DCR-graphs changes on transitions. Where the configuration of an event structure only grows, by adding events each time an event happens, parts of a marking can both increase and decrease during the course of a run, and even loose elements and gain others on the same transition.

One of the possible solutions that we investigated was to see if one could use multisets to simulate an increasing and decreasing marking. By using two multi-sets to simulate a single set from the marking. The idea was for any element that at any time was added to the set of a marking (meaning it was not in it when added through a union operator), we would add this event to the first of the two multisets simulating this set, a multiset that one could say kept track of all elements added to the markings set. Similar for any element that was removed from the markings set at some time, these elements would be added to the second multiset, which would keep track of elements that were removed.

This could be thought of as having the sets PS as the set we simulate from the marking, MS_a as the multiset keeping track of the added elements and MS_r as the multiset keeping track of the removed sets. Where $MS_a \setminus MS_r = PS$.

The strength here is updating any single multi set is done just using the union operator, while allowing for transition labels to show exactly what event is happening.

The problem with this approach were to make sure that after a transition we would get the correct simulation; meaning that for a transition $PS \xrightarrow{\alpha} PS'$ we would be able to have a transition $(MS_a, MS_r) \xrightarrow{\alpha} (MS'_a, MS'_r)$ where $MS'_a \setminus MS'_r = PS'$. To be able to do this we had to know before a transition occurred exactly whatl set we were simulating, in order to be able to add the correct elements to the multisets. The only way to know this in psi-calculi would be to define entailment between conditions and assertions, so that only assertions simulating the specific marking the condition was looking for was entailed. This would mean we would have to check for every marking that would enable an event with their own conditions, for every single event, effectively creating a state machine. State machines have the problem of state explosion which would result in having to make very large psi-calculi processes. A second problem we faced was problems in proving that the assertions we left behind after a transition would have the correct sets, such that when it got composited with the earlier assertions we would get a frame that was simulating the correct marking. When we found a solution that was easier to prove and would give nicer and more compact psi-calculi processes, this idea was shelved as we finished up the solution presented in Chapter 4.

While working on the multi set approach, we were also thinking of using a notion of generations for assertions, which is a nice way to make only the data from the assertions of newest generation to be kept in the composition of assertions. With this we would be able to say that the last made assertion would be the only assertion that gets considered in entailments. As only one assertion would be considered at any time we had to place the entire marking into each assertion, this would effectively remove the problems of having to use multisets in order to get the assertions give the right markings. The hard part was to get the assertions right, as we would still need to know what the assertion was before a transition, in order to know which assertion to leave behind after the transition. Due to the generation aspect who also would have to be updated with each new assertion the state-machine problem from the multiset idea would explode even more as we would have to expand each state to every possible generation. As DCR-graphs are designed to give finite models for infinite behaviour, we would always have to assume we had an infinite behaviour DCR-graph and make infinitely large processes, or only accept those with finite behaviour.

The thought of using a function which would take the current assertion as a parameter in order to make the new one was considered but discarded as the assertions are built of terms. This would have been a good idea as it would have given a compact psi-calculi processes for each event, as well as nice transition labels like we had for event structures.

The solution we got came from moving to communications, where each event would have its condition guarded input. The messages sent around would be the same as the current assertions, and the data received on an input would be used as variables in creating the next output process, and a new assertion with higher generation than the one we received. This is the solution we present in Chapter 4.

1.3 Summary of contributions

This thesis presents the encoding of two true concurrency models into Psi-calculi. These models are the popular prime event structures, and the newer dynamic condition response graphs (DCR-graphs) that is a conservative generalization of event structures. We define two corresponding instantiations which we name eventPsi and dcrPsi respectably. In combination with these instantiations we also define two functions ESPSI and DCRPSI that take an event structure or an DCR-graph respectably, and return psi-calculi processes for their respective instantiations.

We also check that the instantiations maintain certain properties that the models they instantiate have. One of the main properties we are interested in is that we have a correlation between the states of the models and the assertions of our psi-calculi instantiations. Particular we want the assertions in eventPsi to have a correlation with the configurations of event structure, and the assertions in dcrPsi have a correlation with the markings of DCR-graphs.

For eventPsi we obtained a nice instantiation. Here we define an independence relation that is giving the exact same independences in an eventPsi-process that we have in the event structure this process is representing. We define a specific syntax that a eventPsi process must follow to represent a correct event structure. In addition to this we give a definition of refinement functions on eventPsi-processes, that let us refine eventPsi-processes in the same way as we can refine event structures, and get the same process as we would get by creating it from the refined event structure using ESPSI.

For dcrPsi we show that the transitions are correctly showing the same changes in the assertions as we have in the DCR-graphs between markings. But as we have no real refinement of processes and that independence of events is non-trivial to define, we do not have results on these parts.

The work presented in this thesis has been published and/or presented at international workshops. We list these here in chronological order.

(1) A first short article, backed by an extended version available online, was presented by Håkon Normann and published as (extended abstract):

"Event Structures as Psi-calculi". in 25^{th} Nordic Workshop on Programming Theory (NWPT'13), November 2013, Tallinn, Estonia. Editors: Tarmo Uustalu and Juri Vain. 3 pages.

(2) A more mature work has been accepted for publication (and presentation) as:

"Concurrency models with causality and events as psi-calculi". 7^{th} Interaction and Concurrency Experience (ICE2014), June 2014, Berlin Germany. Editors: Ivan Lanese and Ana Sokolova. Electronic proceedings in Theoretical Computer Science. (coauthors: C. Prisacariu and T. Hildebrandt). 15 pages.

(3) The work currently in progress in Chapter 5 has been accepted as a presentation at:

"True Concurrency Semantics for Psi-calculi". 1^{st} International Workshop on Meta Models for Process Languages (MeMo), June 2014, Berlin, Germany. (coauthors: C. Prisacariu and T. Hildebrandt). 6 pages.

Chapter 2

Background

2.1 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, [BJPV11, Sec.2.1] giving an introduction to nominal sets used in psi-calculi. We will give a short presentation of nominal datatypes in this thesis, but for those who want more background on this we refer 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.

2.1.1 Nominal datatypes

A traditional datatype can be build from a signature of constant symbols, functions symbols, etc. A nominal datatype is more general, for example it can also contain binders and identify alpha-variants of terms. Formally a nominal datatype is not required to be build in any particular way; the only requirements are related to the treatment of atomic symbols.

As usual we assume a countably infinite set of atomic *names* $\mathcal{N}$ ranged over by $a, ..., z$. Intuitively, names will represent the symbols that can be statically scoped, and also represent symbols acting as variables in the sense that they can be subjected to substitution.

A nominal set [Pit13] is a set equipped with *name swapping* functions, written (ab) for any names a, b. An intuition is that for any member X it holds that $(ab) \cdot X$ is X with a replaced by b and b replaced by a. Formally, a name swapping is any function satisfying certain natural axioms such as $(ab) \cdot ((ab) \cdot X) = X$. One main point to this is that even though we have not defined any particular syntax one can define what it means for a name to "occur" in an element: it is simply that it can be affected by swappings. The names occurring in this way in an element X constitute the *support* of X, written $n(X)$. We write $a\sharp X$, pronounced "a is fresh for X", for $a \notin n(X)$. In an inductively defined datatype without binders we will have $a\sharp X$ if a does not occur syntactically in X. In for example the lambda calculus where alpha-equivalent terms are identified (i.e. the elements are alpha-equivalence classes of terms) the support corresponds to the free names. If A is a set or a sequence of names we write $A\sharp X$ to mean $a \in A : a\sharp X$

A function f is *equivariant* if $(ab) \cdot f(X) = f((ab) \cdot X)$ holds for all X.

A *nominal datatype* is a nominal set together with a set of equivariant functions on it. In particular psi-calculi consider substitution functions that substitute elements for names. If X is an elements of a datatype, $\tilde{a}$ is a sequence of names with duplicates and $\tilde{Y}$ is an equally long sequence of elements of possibly another datatype, the *substitution* $X[\tilde{a} := \tilde{Y}]$ is an element of the same datatype as X.

2.1.2 Psi-calculi syntax

For psi-calculi it is not needed to define exactly what a substitution does. The only formal requirements are that a substitution is an equivariant function that satisfies two substitution laws:

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

Law 1 says that substitution may not loose names: any name b in the objects $\tilde{T}$ that substitute for names $\tilde{a}$ occurring in X must also appear in the substitution $X[\tilde{a} := \tilde{T}]$.

Law 2 is a form of alpha-conversion for substitutions; here it is implicit that $\tilde{a}$ and $\tilde{b}$ have the same length, and $(\tilde{a}\tilde{b})$ swaps each element of $\tilde{a}$ with the corresponding element of $\tilde{b}$.

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:

T	terms (data/channels)
C	conditions
A	assertions

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

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 $\mathbf{1}$ 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.

Given valid psi-calculus parameters, the psi-calculi *agents*, ranged over by $P, Q, \ldots$, are of the following forms.

$\mathbf{0}$	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 names a inside processes P
$P \mid Q$	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 [AF01], 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 $a \leftrightarrow a$; this use is mimicking the pi-calculus nondeterministic 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.

Replication rule opens up for a possibly infinitely large process. This comes from the fact that any transition within the replication process, will lead to a reduced process in parallel composition with the replicating process that remains unreduced.

2.1.3 Examples of psi-calculi instances

Example 2.1.1 (pi-calculus as an instance) *To obtain pi-calculus [MPW92] as an instance of psi-calculus use the following, built over a single set of names $\mathcal{N}$:*

$$\begin{array}{lcl}
\mathbf{T} & \triangleq & \mathcal{N} \\
\mathbf{C} & \triangleq & \{a = a \mid a, b \in \mathbf{T}\} \\
\mathbf{A} & \triangleq & \{\mathbf{1}\} \\
\dot{\leftrightarrow} & \triangleq & = \\
\otimes & \triangleq & \lambda\Psi_1, \Psi_2.\mathbf{1} \\
\mathbf{1} & \triangleq & 1 \\
\vdash & \triangleq & \{(\mathbf{1}, a = a) \mid a \in \mathbf{T}\}
\end{array}$$

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 2.1.2 *From the instance created in Example 2.1.1 one can obtain the polyadic pi-calculus by adding the 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 and tuples also in subject position in agents, but those do not give rise to any transitions, since in this psi-calculus $M \dot{\leftrightarrow} M$ is only entailed when M is a name, i.e., only names are channels.*

2.1.4 Operational semantics for psi-calculi

Psi-calculus is given an operational semantics in [BJPV11] using labelled transition systems, where the nodes 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 synchronization/communication action:

The *actions* ranged over by α, β are of the following three kinds:

$\overline{M}\langle(\nu\tilde{a})N\rangle$	Output, where $\tilde{a} \subseteq n(N)$
$\underline{M}\langle N\rangle$	Input
τ	Silent

For actions we refer to M as the *subject* and N as the *object*.

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 context assertion is obtained using the notion of *frame* which essentially collects (using the composition operation) the outer-most assertions of a process. The frame $\mathcal{F}(P)$ is defined inductively on the structure of the process as:

$$\begin{aligned}
&\mathcal{F}((\!|\Psi|\!)) = \Psi\\
&\mathcal{F}(P \,|\, Q) = \mathcal{F}(P) \otimes \mathcal{F}(Q)\\
&\mathcal{F}((\nu a)P) = (\nu a)\mathcal{F}(P)\\
&\mathcal{F}(!P) = \mathcal{F}(\mathbf{case}\ \tilde{\varphi} : \tilde{P}\,) = \mathcal{F}(\overline{M}\langle N\rangle.P) = \mathcal{F}(\underline{M}\langle(\lambda\tilde{x})N\rangle.P) = \mathcal{F}(0) = \mathbf{1}
\end{aligned}$$

For a simple example, if $a\sharp\Psi_1$:

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

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 $\mathbf{1}$

We give the transition rules for psi-calculus, as defined in [BJPV11, Table 1]. The (CASE) rule shows how the conditions are tested against the context assertions. 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 processes.

$$\frac{\Psi \vdash M \leftrightarrow K}{\Psi \rhd \underline{M}(\lambda\tilde{y})N.P \xrightarrow{\underline{K}N[\tilde{y}:=\tilde{L}]} P[\tilde{y} := \tilde{L}]}\ (\text{IN})$$

$$\frac{\Psi \vdash M \leftrightarrow K}{\Psi \rhd \overline{M}N.P \xrightarrow{\overline{K}N} P}\ (\text{OUT})$$

$$\frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \tilde{\varphi} : \tilde{P} \xrightarrow{\alpha} P'}\ (\text{CASE})$$

$$\frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{M}(\nu\tilde{a})N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{K}N} Q' \qquad \Psi_Q \otimes \Psi_P \otimes \Psi \vdash M \leftrightarrow K}{\Psi \rhd P \,|\, Q \xrightarrow{\tau} (\nu\tilde{a})(P' \,|\, Q')}\ (\text{COM})$$

$$\frac{\Psi \otimes \Psi_Q \rhd P \xrightarrow{\alpha} P' \qquad bn(\alpha)\#Q}{\Psi \rhd P \,|\, Q \xrightarrow{\alpha} P' \,|\, Q}\ (\text{PAR})$$

$$\frac{\Psi \rhd P \xrightarrow{\alpha} P' \qquad b\sharp\alpha, \Psi}{\Psi \rhd (\nu b)P \xrightarrow{\alpha} (\nu b)P'}\ (\text{SCOPE})$$

$$\frac{\Psi \rhd P \xrightarrow{\overline{M}(\nu\tilde{a})N} P' \qquad b\sharp\tilde{a}, \Psi, M \qquad b \in n(N)}{\Psi \rhd (\nu b)P \xrightarrow{\overline{M}(\nu\tilde{a}\cup\{b\})N} P'}\ (\text{OPEN})$$

$$\frac{\Psi \rhd P|!P \xrightarrow{\alpha} P'}{\Psi \rhd !P \xrightarrow{\alpha} P'}\ (\text{REP})$$

The (IN)-rule says that any input can happen for a channel M if it is channel equivalent with the channel K that will show in its transition label.

The (OUT) rule says more or less the same as the (IN)-rule in that the channel of the process and the channel of the transition label must be channel equivalent.

The (CASE)-rule says that if one of the possible case processes can do an α transition into P', and if the condition is entailed by the assertion over the entire case process, then the entire case process can do an α transition to P'.

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 $\mathcal{F}(Q) = (\nu\tilde{b}_Q)\Psi_Q$. We also see that for a synchronization to happen the channel in the transition labels for the input and output processes must be channel equivalent.

In the (PAR)-rule the assertion Ψ_Q comes from the frame of $\mathcal{F}(Q) = (\nu\tilde{b}_Q)\Psi_Q$, and say that if P can do an α transition to P' and α is a fresh name in Q then the parallel composition $P \mid Q$ can do an α transition to $P' \mid Q$.

The (SCOPE) says that for a process $(\nu b)P$ that is under the scope of b can have an α transition to $(\nu b)P'$ if P can have an α transition to P' and b is fresh in both α and the assertion that the process is under.

The (REP) rule says that it can have a transition from $!P$ to P' if it could have the same transition from $P \mid !P$ to P' meaning if P could transition we generate a copy of this in parallel composition with $!P$ and have the transition on P.

The symmetric versions of (COM) and (PAR)-rules are elided. In the rule (COM) it is assumed that $\mathcal{F}(P) = (\nu\tilde{b}_P)\Psi_P$ and $\mathcal{F}(Q) = (\nu\tilde{b}_Q)\Psi_Q$ where $\tilde{b}_P$ is fresh for all of $\Psi, \tilde{b}_Q, Q, M$ and P, and that $\tilde{b}_Q$ is correspondingly fresh. In the rule PAR it is assumed that $\mathcal{F}(Q) = (\nu\tilde{b}_Q)\Psi_Q$ where $\tilde{b}_Q$ is fresh for Ψ, P and α. In OPEN the expression $\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

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.

2.1.5 Bisimilarity in psi-calculi

For the standard pi-calculi the notion of strong bisimulation is used to formalise when two agents "behave the same way"; it is defined as a symmetric binary relation $\mathcal{R}$ satisfying the simulation propert: $\mathcal{R}(P, Q)$ implies that for α such that $bn(\alpha)\sharp Q$,

$$\text{if } P \xrightarrow{\alpha} P' \text{ then } Q \xrightarrow{\alpha} Q' \wedge \mathcal{R}(P', Q').$$

But psi-calculi need to take the assertions into consideration. The behaviour of an agent is always taken with respect to an environmental assertion. For psi-calculi bisimulation is defined as a ternary relation $\mathcal{R}(\Psi, P, Q)$, saying that P and Q behave in the same way when the enviroment asserts Ψ. Because of this two additional issues arise. The first is that the agents can affect their environment through their frames (and not only by performing actions), and this must be represented in the definition of bisimulation. The second is that the environment (represented by Ψ in $\mathcal{R}(\Psi, P, Q)$) can change, and for P and Q to be bisimilar they must continue to be related after such changes. This leads to the following strong bisimulation.

Definition 2.1.3 A bisimulation $\mathcal{R}$ *is a ternary relation between assertions and pairs of agents such that* Ψ *in* $\mathcal{R}(\Psi, P, Q)$ *implies all of*

1. *Static Equivalence:* $\Psi \otimes \mathcal{F}(P) \simeq \Psi \otimes \mathcal{F}(Q)$

2. *Symmetry:* $\mathcal{R}(\Psi, Q, P)$

3. *Extension of arbitrary assertions:* $\forall \Psi'.\mathcal{R}(\Psi \otimes \Psi', P, Q)$

4. *Simulation: for all* α, P' *such that* $bn(\alpha)\sharp\Psi, Q$ *there exists a* Q' *such that*

$$\textit{if } \Psi \rhd P \xrightarrow{\alpha} P' \textit{ then } \Psi \rhd Q \xrightarrow{\alpha} Q' \wedge \mathcal{R}(\Psi, P', Q')$$

$P \overset{\bullet}{\sim}_\Psi Q$ *is defined to mean that there exists a bisimulation* $\mathcal{R}$ *such that* $\mathcal{R}(\Psi, P, Q)$*, and* $\overset{\bullet}{\sim}$ *is written for* $\overset{\bullet}{\sim}_1$

2.2 On event structures

For event structures we try to follow the standard notation and terminology from [WN95, sec.8]. We choose to use prime event structures as this is one of the most basic and well known models of concurrency. Moreover, the DCR-graph model described in Section 2.5 which is our final goal, is a conservative extension of event structures. Therefore, a first step is to look at event structures.

Definition 2.2.1 (prime event structures) A labelled prime event structure *over alphabet Act is a tuple* $\mathcal{E} = (E, \leq, \sharp, l)$ *where* E *is a 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.,* $\forall d, e, f \in E : d \leq e \wedge d\sharp f \Rightarrow e\sharp f.$

and $l : E \rightarrow Act$ *is the labelling function.*

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 of the system.

Definition 2.2.2 (concurrency) Casual independence (or concurrency) *between events is defined in terms of the above two relations as*

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

capturing the intuition that two events are concurrent when they are not in conflict and there is no causal dependence between the two.

Example 2.2.3 *We make a simple ES with four events* a, b, c, d *where we have the relations* $a \leq c, b \leq d, b\sharp c, b\sharp c, d\sharp a, a\sharp d, c\sharp d, d\sharp c$*. This would give us the tuple* $ES = (\{a, b, c, d\}, \{(a, c), (b, d)\}, \{(b, c), (c, b), (d, a), (a, d), (c, d), (d, c)\}, l)$.

The usual graphical representation of this system is pictured in Figure 2.1. On the figure we have not added the conflict relation between c *and* d*, because this is deducible from the conflict heredity requirement in Def 2.2.1.2. We have that* $a||b$ *for this example.*

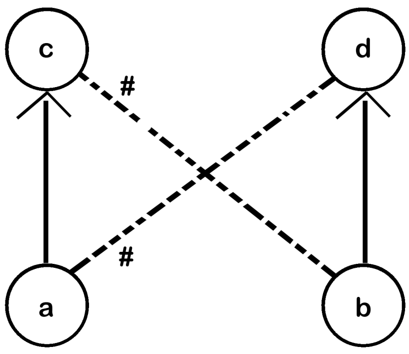

Figure 2.1: Graphical view of an ES

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

Definition 2.2.4 (configurations) *Define a* configuration *of an event structure $\mathcal{E} = (E, \leq, \sharp)$ to be a 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 $\mathcal{C}_{\mathcal{E}}$.

Note in particular that $\emptyset$ is a configuration of any event structure(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 $C' = C \cup \{e\}$.

Remark 2.2.5 *It is known (see eg., [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 $\mathcal{C}_{\mathcal{E}}$ as follows:*

1. *$e \leq e'$ iff $\forall C \in \mathcal{C}_{\mathcal{E}} : e' \in C \Rightarrow e \in C$;*

2. *$e \sharp e'$ iff $\forall C \in \mathcal{C}_{\mathcal{E}} : \neg(e \in C \wedge e' \in C)$;*

3. *$e || e'$ iff $\exists C, C' \in \mathcal{C}_{\mathcal{E}} : e \in C \wedge e' \notin C \wedge e' \in C' \wedge e \notin C' \wedge C \cup C' \in \mathcal{C}_{\mathcal{E}}$.*

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 in sync with similar notations we use in this thesis for similar sets defined for DCRs too), and $\sharp e = \{e' \in E \mid e' \sharp e\}$ those events e is in conflict with.

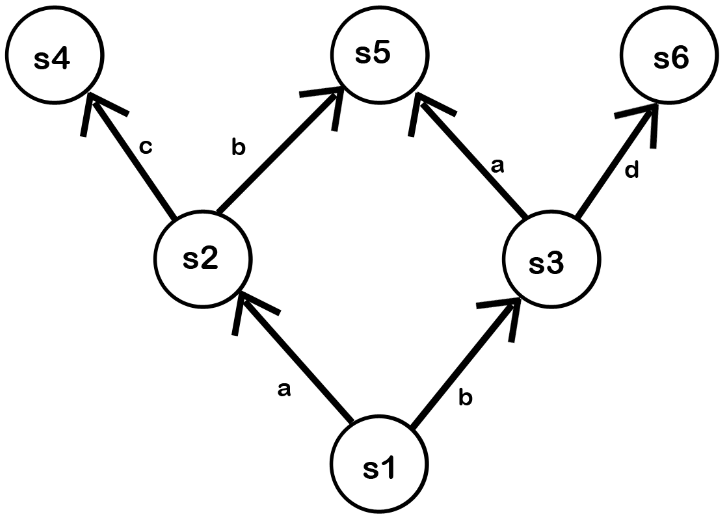

Figure 2.2: Transition graph of an ES

Example 2.2.6 *Let us take the event structure we made in Example 2.2.3 called ES. For ES we are able to make a transition system between the configurations or states of ES to be like the transition system graph in Figure 2.2. We have the possible configurations of ES marked as $s1, s2, s3, s4, s5, s6$, where $s1 = \emptyset$ as no events have happened at the start of the execution. From the transition $s1 \xrightarrow{a} s2$ we have $s2 = s1 \cup \{a\} = \{a\}$. In a similar way we have that the rest of the configurations are $s3 = \{b\}, s4 = \{a, c\}, s5 = \{a, b\}, s6 = \{b, d\}$.*

From these configurations we can find what the relations of ES are, using Remark 2.2.5. We have that the only configuration where c is in is $s4$ and as a is in this configuration, a is in all configurations that c is in so $a \leq c$. Similar we have that d is only in $s6$ and as b is in this configuration, we have $b \leq d$. These are the only two condition relations that we have in ES. For the conflict relations we can see that a is in $s2, s4, s5$ and that d is in none of these, which give us that $a\sharp d$. We see that d is only in $s6$ and as $a \notin s6 \wedge c \notin s6$ we have the relations $d\sharp a, d\sharp c$. c is only found in $s4$ and as we have $b \notin s4 \wedge d \notin s4$ we get relations $c\sharp b, c\sharp d$. b is in the three configurations $s3, s5, s6$ and as we have that c is not in any of them we have $b\sharp c$. This gives us all six conflict pairs we have in ES.

For the final relation we see that $a \in s2$ while $b \notin s2$, and we have that $b \in s3$ while $a \notin s3$. We know that $s2 = \{a\}$ and $s3 = \{b\}$ with $s2 \cup s3 = \{a\} \cup \{b\} = \{a, b\}$ as $\{a, b\} = s4$ we have that $a||b$.

2.3 Refinement of event structures

An action can be considered as a conceptual entity at a chosen level of abstraction. This allows for the representation of systems in a hierarchical way, changing the level of abstraction by interpreting actions on a higher level by more complicated processes on a lower level. This method of give more detail to an action, thus turning the abstract model into a more concrete one, is referred to as refinement of actions or action refinement [vGG01].

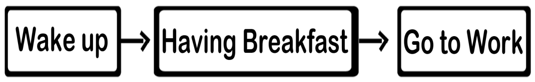

Figure 2.3: Having breakfast event structure

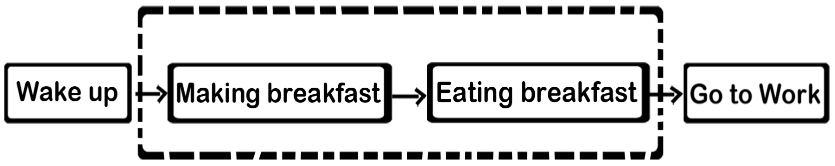

Figure 2.4: refinement by a sequential event structure

Example 2.3.1 *We can look at the actions of waking up, having breakfast and going to work, like visualized in the event structure in Figure 2.3.*

We might want to know a bit more about how one is having breakfast, this can be done by refining the action of having breakfast into several other actions. We first go to the abstract level where having breakfast is build up of the actions make breakfast and eat breakfast. Make breakfast must happen before eat breakfast, as otherwise there would be nothing to eat. This is visualized in the Figure 2.4.

We can go even further down the levels of abstraction and look at what we are making for breakfast. This can be that we fry veggie bacon and fry an egg. These two actions can happen at the same time (no one says you can't be using the same frying pan for both the bacon and the egg at the same time), so we will not have any internal relations in the event structure we are building. This new event structure after refinement a second time looks like in Figure 2.5

If we are even more interested in how one is having breakfast, one can refine the eating breakfast action into actions that say where one is eating breakfast. This can be eating the breakfast in the kitchen or in the living room. But one can not eat breakfast in more than one room (no walk around), so these two events must be in conflict. Trying to refine "eat breakfast" event with these two new events, we will come into a situation where our morning bird would not be able to go to work, due to the conflict heredity. Refining this way would result in the event structure in Figure 2.6.

We see from the previous example that by refining an event into an event structure with conflicts, one can cause other events that could happen before we refined, to no longer be able

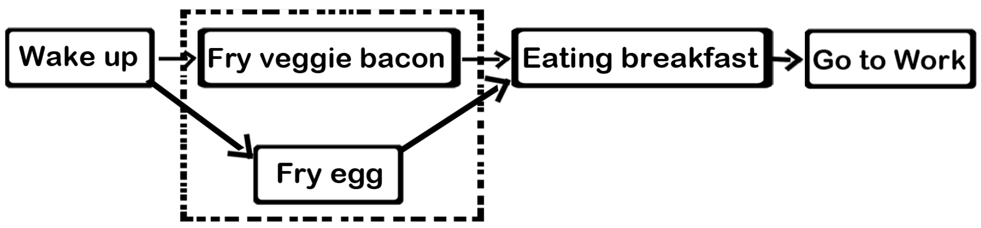

Figure 2.5: refinement by a parallel event structure

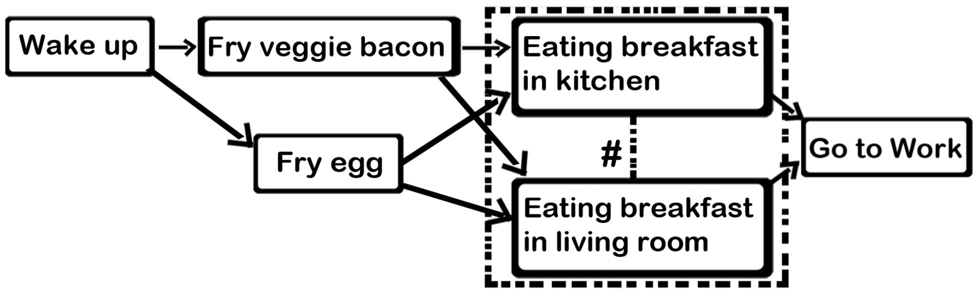

Figure 2.6: refinement by a conflict event structure

to execute. Depending on how one sees the correct refinement of events this may be a problem or not in terms of correctness. The way we are presenting here removes the option of getting into these situations, as we demand the event structure we refine an event into, to be conflict-free. In the above example one can easily see that getting into an always conflicted situation is not a preferred one.

Actions are often considered single entities which when happening give no information about how long the action took (often thought of as instantaneous), or what making it happen entails. This means that the behaviour of the original system does not have any information about the system we are refining it into. There is though the idea where we want the event structure we get to preserve the behaviour of the one we refined from.

The intuition of refinement is to take one action (which is thought as an abstraction) and give it more structure. Since the same action can be instantiated several times at different points in the system, i.e., by different events, all these events labelled by the same action are being given more structure by replacing them with a new event structure. For example one event can become a sequence of events, or the parallel composition of deterministic components.

But refinement is restricted to not contain conflicts, i.e., not contain choices. This is because of technical reasons that make it not possible to define the new conflict relation in a way that we obtain prime event structures after refinement; but also there are natural counter examples for requiring conflict-free refining event structure, like seen in Example 2.3.1 or in [vGG01].

We take the definition of action refinement from [vGG01]. This is done using a function ref that is a function from actions to finite, non-empty event structures without conflict (i.e., the conflict relation is empty). This is considered as a given function to be used in the refinement operation. This refinement operation can be also seen as a function from event structures together with functions as above, and returning new event structures; it is like an algorithm. For notation economy this algorithm is also denoted by ref, to connect it with the essential input it takes as the refinement function $ref : Act \to ES$ (conflict-free).

Definition 2.3.2 (refinement for prime event structures) *For an event structure $\mathcal{E}$ with events labelled by $l : E \to Act$ with actions from Act we have the following definitions.*

(i) A Function $ref : Act \to \boldsymbol{E}_{prime}$ 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.*

(ii) Let $\mathcal{E} \in \boldsymbol{E}_{prime}$ and let ref be a refinement function.

Then $ref(\mathcal{E})$ is the prime event structure defined by:

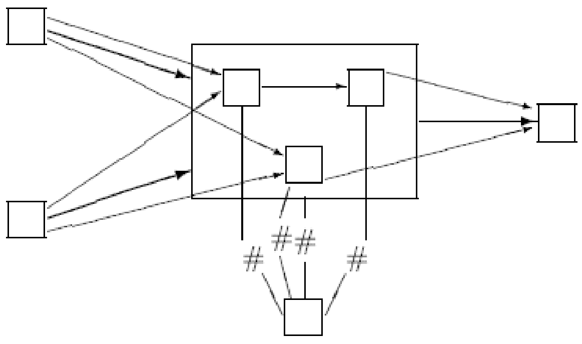

Figure 2.7: Graphical idea of refinement of events

- $E_{ref(\mathcal{E})} := \{(e, e') | e \in E_{\mathcal{E}}, e' \in E_{ref(l_{\mathcal{E}}(e))}\}$,
- $(d, d') \leq_{ref(\mathcal{E})} (e, e')$ *iff* $d \leq_{\mathcal{E}} e$ *or* $(d = e \wedge d' \leq_{ref(l_{\mathcal{E}}(d))} e')$,
- $(d, d') \sharp_{ref(\mathcal{E})} (e, e')$ *iff* $d \sharp_{\mathcal{E}} e$,
- $l_{ref(\mathcal{E})}(e, e') := l_{ref(l_{\mathcal{E}}(e))}(e')$.

The refinement operation ref takes any event in ES (the parent part), and pairs its name with the name of each of the events it refines into (the child part). The new conflict relation between the pair of names is determined by the conflict relations between the parent events, i.e. if the parent events were in conflict, then the new pair is in conflict. The new condition relations are either between pairs where the parents were in a condition relation, or where the parents are the same event, and the children were in a condition relation.

Example 2.3.3 *We continue with our event structure ES from Example 2.2.3. We have a function ref which will refine the action a into the event structure $(\{e, f\}, \{(e, f)\}, \emptyset, l_{ref(a)})$ and let the other three events stay the same.*

For simplicity in writing $ref(b) = b$ instead of (b, b) for the refined action of b and similar for the actions c and d. We have that $ref(a) = \{(a, e), (a, f)\}$ as $E_{ref(l_{\mathcal{E}}(a))} = \{e, f\}$. The condition relations we are getting are $(a, e) \leq_{ref(ES)} (a, f)$ due to $a = a \wedge e \leq_{ref(l_{ES}(a)} f$, and $(a, e) \leq_{ref(ES)} c, (a, f) \leq_{ref(ES)} c$, from $a \leq c$. We also maintain $b \leq d$ relation as both actions here were replaced with them selves. From the relations $a \sharp d, d \sharp a$ we get the four relations $(a, e) \sharp_{ref(ES)} d, (a, f) \sharp_{ref(ES)} d, d \sharp_{ref(ES)} (a, e), d \sharp_{ref(ES)} (a, f)$.

This gives the event-structure
$ref(ES) = (\{(a, e), (a, f), b, c, d\}, \{((a, e), c), ((a, f), c), (b, d)\},$
$\{((a, e), d), ((a, f), d), (d, c), (b, c)\})$.

On a graphical presentation we would end up with an event structure graph similar to Figure 2.8

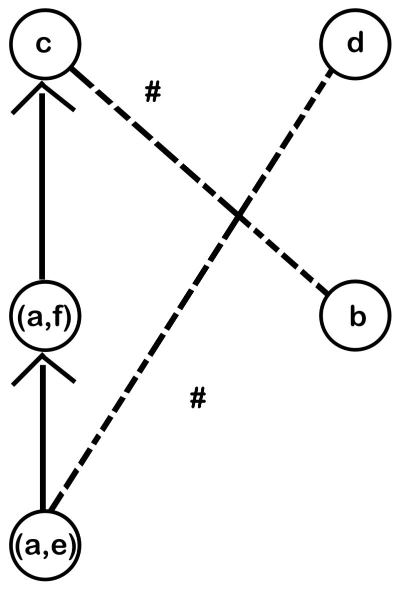

Figure 2.8: Refined ES as graph

2.4 On Condition Response Event Structures

Condition Response Event Structures (CRES) presented in [HM10b] is an intermediate step from Event Structures to Dynamic Condition Response Graphs. CRES generalize event structures by introducing the notion of progress based on a response relation. CRES is interesting in its own right as an extensional event-based model with progress, abstracting away from the intentional representation of repeated behaviour. In particular it allows for an elegant characterization of weakly fair runs of event structures.

Generalising a prime event structure to CRES is done by adding the *response* relation $\bullet\!\!\rightarrow$, which can be seen as the dual to the condition relation. the set $\{e'|e \bullet\!\!\rightarrow e'\}$ represents the events that must happen (or be in conflict) after event e has happened for the run to be accepting. The resulting structure, named *condition response event structures*, in this way adds the possibility of state progress conditions. A subset Re is also introduced, capturing the *initial* responses, which are the events that are initially required to eventually happen (or become in conflict). In this way the structure can represent the state after an event has been executed. As shown bellow, it also allows to capture the notion of maximal runs.

Definition 2.4.1 *A labelled condition response event structures (CRES) over an alphabet* Act *is a tuple* $(E, Re, Act, \bullet\!\!\rightarrow, \rightarrow\!\!\bullet, \sharp, l)$ *where*

1. $(E, \rightarrow\!\!\bullet, \sharp, l)$ *is a labelled prime event structure, referred to as the underlying event structure*
2. $\bullet\!\!\rightarrow \subseteq E \times E$ *is the* response *relation between events, satisfying that* $\rightarrow\!\!\bullet \cup \bullet\!\!\rightarrow$ *is acyclic*
3. $Re \subseteq E$ *is the set of* initial responses.

A configuration c *and run* ρ *of a CRES is respectively a configuration and run of the underlying event structure. We define a run* $(e_0, l(e_0)), (e_1, l(e_1)), ...$ *to be accepting if* $\forall e \in E, i \geq 0.e_i \bullet\!\!\rightarrow$

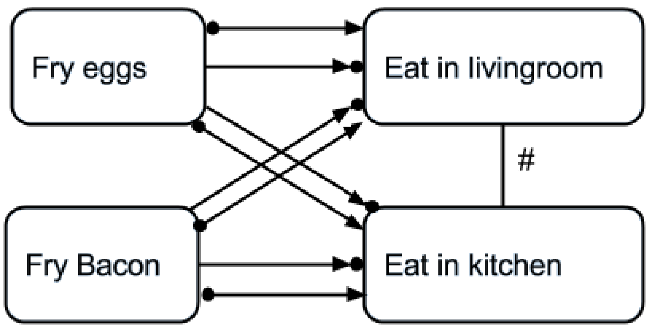

Figure 2.9: Example of simple CRES

$e \Rightarrow \exists j \geq 0.(e\sharp e_j \vee (i \leq j \wedge e = e_j))$ *and* $\forall e \in R.\exists j \geq 0.(e\sharp e_j \vee e = e_j)$. *In words, any pending response event must eventually happen or be in conflict.*

A prime event structure can be regarded as a condition response event structure with empty response relation. This provides an embedding of prime event structures into condition response event structures that preserves configurations and runs.

Proposition 2.4.2 *The labelled prime event structure* $(E, Act, \leq, \sharp, l)$ *has the same runs as the CRES* $(E, \emptyset, Act, \leq, \emptyset, \sharp, l)$ *for which all runs are accepting.*

We can also embed event structures into CRES by considering every condition to also be a response, and all events with no condition to be initial responses. This characterizes the interpretation in [Che95] where only *maximal* runs are accepting. In other words, the embedding captures the notion of weakly fair execution of event structures.

Proposition 2.4.3 *The labelled prime event structure* $(E, Act, \leq, \sharp, l)$ *has the same runs and maximal runs as respectively the runs and accepting runs of the CRES* $(E, \{e|e \downarrow = \emptyset\}, Act, \leq, \leq, \sharp, l)$.

Example 2.4.4 *We can take the event structure where we have the events: "Fry bacon", "Fry eggs", "Eat in living room", "Eat in kitchen". Here we have that both fry events are conditions for the eat events, as we need to make what we want to eat. Since we can only eat in one place the eat events are in conflict with each other.*

Taking this event structure we can make a CRES where we either have no initial responses (we do not necessary need to eat anything), or we have that both eat events are initial responses (we need something to eat). For both ways we add response relations between events that already have a condition relation. This says that if we start making food we need to eat it also. Both CRES's are giving us the same graphical look if we add the same relations to them looking like Figure 2.9

These two CRES's may look the same but due to the fact that only one has initial responses we will get two different set's of accepting runs. Where the first one only will get accepting runs when one of the eat events has happened, the second one can have an accepting run where no events happen at all, in addition to those the first CRES has.

2.5 On DCR-graphs

Dynamic Condition Response Graphs (DCR-graphs) is a recent model of concurrency, that was originally presented in [HM10b]. This model focuses on giving a finite representation of possibly infinite behaviour, while also facilitating the notion of accepting runs. Particular to DCR-graphs is the fact that a single event may constantly switch between being enabled or not being enabled during a run. This is because that any enabled event may execute, and by executing, change the state of the DCR-graph, where some events that was not enabled now is enabled, and some that was enabled no longer is enabled. An effect of this is that DCR-graphs are very well suited for modelling work-flows, especially those where you do not know in advance how many times a single event may have to happen in order to get it done correctly, without having to generate several instances of the same event, one for each possible time it may happen. In addition to this, DCR-graphs have a graphical notation that makes it easy for people in industry to see what a DCR-graph will do and what work-flow it is modelling.

Example 2.5.1 *An example of a DCR-graph can be created based on the TV-show Hell's Kitchen (or any other restaurant kitchen, where several cooks are making the food, and one head-chef decides after checking if it is good enough for the customers), is the work-flow for one cook, based on a single order. In advance we do not know how many times chef Ramsay will demand the food to be re-fired, because the food he gets up to the pass is not perfect.*

An event structure built to model this situation may have to be infinite large in order to capture the fact that the food may never get accepted and the cook has to remake it forever all night long (this is not directly true as the restaurant do not have infinite amounts of ingredients, and in the end either Ramsay gets so angry he shuts kitchen down or the customer leaves hungry, but these are possibilities we do not consider in this example). Similar we would be able to get an infinite CRES model for same reasons, though we would be able to say that we have an accepting run once the food finally gets accepted.

In DCR-graphs on other hand we can get away with only having to have five events. These events will be "Get order", "Cook order", "Take to pass", "Re-fire demanded" and "Food accepted". As we are modelling single orders, we can only get an order in once so we make that the "get order" event can only be enabled until it execute. It is intuitive that the cook has to make the food before being able to take it to the pass, and we should not cook anything unless we are ordered to do so through either getting an order in or having to re-fire the food. If the food is accepted we end the execution as the order has been fulfilled, but if the food is rejected then a re-fire is ordered and we need to cook the food again before taking it to the pass. This DCR can be represented by the graph seen in Figure 2.10. This picture was made using the graphical modelling tool for DCR-graphs developed by Exformatics (www.exformatics.com) that can be found at www.itu.dk/research/models/wiki/index.php/DCR_graphs_editor.

We will follow the notations of DCR's from [HM10a, HMS12] for the work we do.

Definition 2.5.2 (DCR Graphs) *We define a* Dynamic Condition Response Graph *to be a tuple* $G = (E, M, \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$ *where*

1. *E is a set of events,*

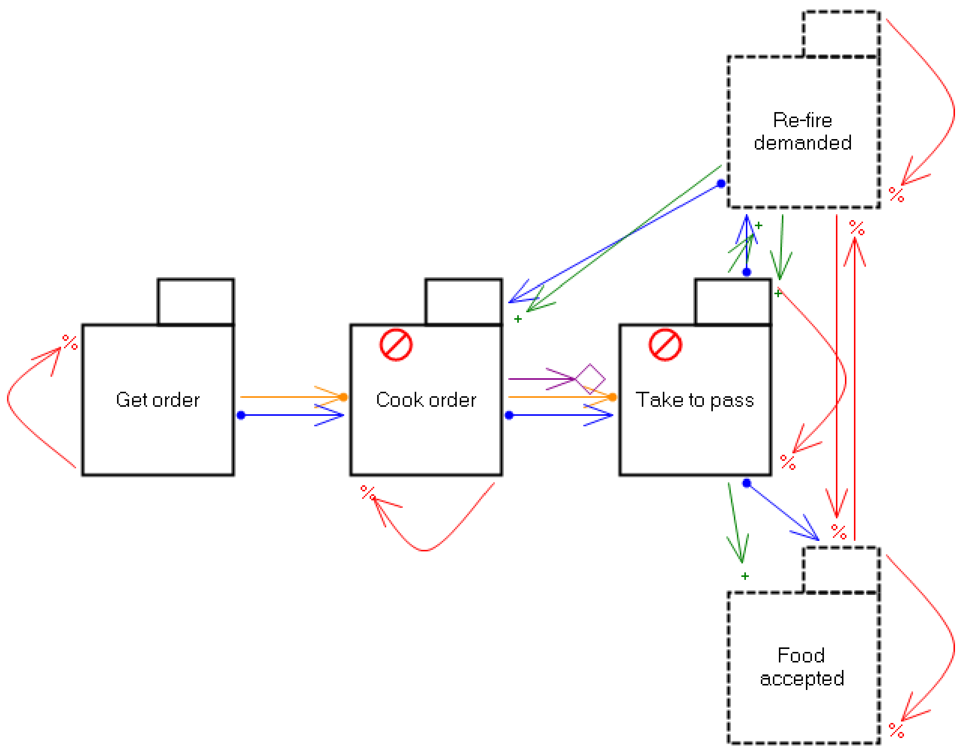

Figure 2.10: DCR of how a line cook has to handle an order

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 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 $\{e' \in E \mid e' \rightarrow e\}$ of events $e' \in E$ which are in the respective relation with e.

A marking $M = (Ex, Re, In)$ represents a state of the DCR. One should understand Ex as the set of executed events, Re the set of events that must happen sometime in the future, and In the set of included events, i.e., those that *can* happen in the next steps. The five relations impose constraints on the events and dictate the dynamic inclusion and exclusion of events.

For a DCR Graph $(E, M, \rightarrow\!\bullet,\ \bullet\!\rightarrow, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%)$ and a marking $M = (Ex, Re, In)$, we say that an event $e \in E$ is enabled in M, written $M \vdash e$, iff $e \in In \wedge (In \cap \rightarrow\!\bullet e) \subseteq Ex \wedge (In \cap \rightarrow\!\diamond e) \subseteq E \backslash Re$. Intuitively an event can only happen if it is included, all it's included preconditions have been executed, and none of the included events that are milestones for it are scheduled responses. The behaviour of a DCR is given through transitions between markings done by executing enabled events. The result of the execution of the event e in marking $M = (Ex, Re, In)$ is defined as the new marking $M' \overset{def}{=} (Ex \cup \{e\}, (Re \setminus \{e\}) \cup e\bullet\!\!\rightarrow,$ $(In \setminus e \rightarrow\!\%) \cup e \rightarrow\!+)$. We denote a transition as $M \xrightarrow{e} M'$.

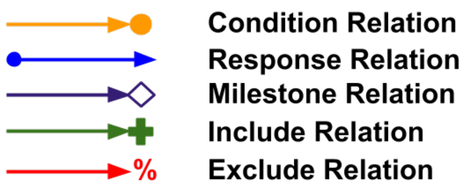

Figure 2.11: Index of DCR relations

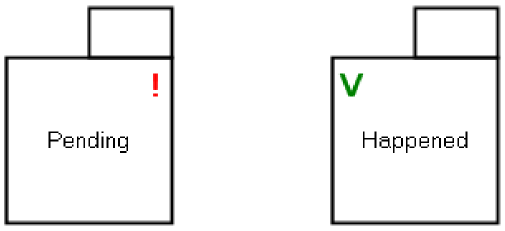

Figure 2.12: Signs for pending and executed events in DCR

Graphically events are drawn as boxes with relations between events being shown as arrows of shape like the formal relations (for better visual looks they have different colours also). These relations are shown in Figure 2.11. The events that are included are shown with a full border, while the excluded events have a dotted border. The graphical representation of DCR-graphs i use as examples here are all made with a graphical modelling tool that can be obtained from www.itu.dk/research/models/wiki/index.php/DCR_graphs_editor. This tool gives the ability to make DCR-graphs and then simulate the runs they can have. In addition to the standard graphical representation, this tool gives additional three graphic symbols that make the graphs easier to understand from just looking at them. The first symbol we can see in Figure 2.10 where it shows a stop sign on the events "cook order" and "take to pass". This sign means that events that have it are not enabled in the current marking and are unable to execute. In Figure 2.12 we see the last two signs we can get. Here we have the red exclamation mark saying that this event is pending an response, and the green V saying that this event has happened at least once already during this run. With these last two symbols one can easily visualize the marking that the DCR-graph is in, where all events with a green V are in the set of executed events, all event with the red exclamation mark are in the set of pending responses, and all events with a full boarder are in the set of included events. The stop sign is not directly needed to see the state of the graph, but it gives a faster way to read which events are enabled and which is not than by having to check its relations to other events and the state they are in.

From the fact that it is not tested against if an event has been executed yet, when determining whether the same event is enabled. We have that an event may be enabled and disabled at several times during a run, even after it has been executed, and its not said it gets disabled when it execute. This gives us that an event may be enabled infinitely often, and able to execute infinitely often during a run. The only way to make sure an event is only able to execute once,

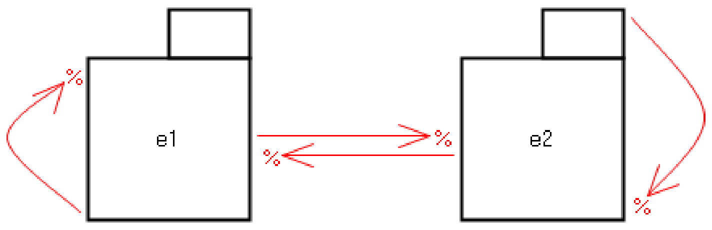

Figure 2.13: conflict relation in DCR

would be to make it exclude itself and make sure it does not get included.

Any CRES can be embedded into a DCR-graph. This is done by leaving the include and milestone relations empty, as these are two additions to DCR graphs which directly expands on CRES. All events are made to exclude themselves, so that no event can be enabled after it has happened once. The conflict relation just excludes all related events, like shown in Figure 2.13. The condition and response relations are the same from CRES.

Example 2.5.3 *We start by taking our graphically defined DCR-graph from Example 2.5.1. Here we have five events that for ease of writing and reading we will rename like this: "Get order" = go, "Cook order" = co, "Take to pass" = tp, "Re-fire demanded" = rd and "Food accepted" = fa. Looking at the graph we can build up the formal DCR-graph tuple. The set of events is $E = \{go, co, tp, rd, fa\}$.*

For the marking we can see that we have no executed events so $Ex = \emptyset$, we have no pending responses so $Re = \emptyset$, and we have only three events that have full borders, i.e. are included $In = \{go, co, tp\}$. This gives us the marking $M = (\emptyset, \emptyset, \{go, co, tp\})$.

The relations can be created from looking at the relations arrows, and make the relation pairs based on them (using for pairing the (from,to) placement in the pairs). This gives us the relations:

$\rightarrow\!\!\bullet = \{(go, co), (co, tp)\}$,

$\bullet\!\!\rightarrow = \{(go, co), (co, tp), (tp, fa), (tp, rd), (rd, co)\}$,

$\rightarrow\!\!\diamond = \{(co, tp)\}$,

$\rightarrow\!\!+ = \{(tp, fa), (tp, rd), (rd, co), (rd, tp)\}$,

$\rightarrow\!\!\% = \{(go, go), (co, co), (tp, tp), (rd, rd), (fa, fa), (fa, rd), (rd, fa)\}$.

For the labels L we let them be the same as the names of the events E and l labels the events with their own names. We thus get the DCR-graph $G = (E, M, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%, L, l)$

Based on the relations we can determine that only go is enabled, both co and tp fail on the part $(In \cap \bullet\!\!\rightarrow_e) \subseteq \emptyset$ due to fact that for co, go is in both $\bullet\!\!\rightarrow_{co}$ and In, the same problem is for tp as co is in both $\bullet\!\!\rightarrow_{tp}$ and In. For the events rd and fa neither of them is in In and thus not enabled. For go we have that it is in the set In and as $\rightarrow\!\!\bullet_{go} = \emptyset$ and $\rightarrow\!\!\diamond_{go} = \emptyset$ we have that it is enabled.

From the relations we know that $go \bullet\!\!\rightarrow = \{co\}$, $go \rightarrow\!\!+ = \emptyset$ *and* $go \rightarrow\!\!\% = \{go\}$.

A transition $M \xrightarrow{go} M'$ *would cause the new marking to change into* $M' = (Ex \cup \{go\}, (Re \setminus \{go\}) \cup go \bullet\!\!\rightarrow, (In \setminus go \rightarrow\!\!\%) \cup go \rightarrow\!\!+) = (\emptyset \cup \{go\}, (\emptyset \setminus \{go\}) \cup \{co\}, (\{go, co, tp\} \setminus \{go\}) \cup \emptyset) = (\{go\}, \{co\}, \{co, tp\})$.

This can also be seen as a transition between DCR-graphs $G \xrightarrow{go} G'$ *where* $G' = (E, M', \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%, L, l)$.

Chapter 3

Encoding event structures in psi-calculi

3.1 The encoding

Due to their popularity, we have chosen to work with a version of event structures called *prime*. These have many nice features like correlations with domains which makes them a good candidate for being used for semantics of concurrent programs. Nevertheless, we believe that other, more general, versions of event structures, like those from [Win86] or [vGP09], can be encoded in psi-calculi following similar ideas as we give here.

Definition 3.1.1 (event psi-calculus) *We define a psi-calculus instance, which we call* eventPsi*, parametrized by a nominal set E, to be understood as* events*, by providing the following definitions of the key elements of a psi-calculus instance:*

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

$$\mathbf{C} \overset{def}{=} 2^E \times 2^E$$

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

$$\dot{\leftrightarrow} \overset{def}{=} =$$

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

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

$$\vdash \overset{def}{=} \Psi \vdash \varphi \text{ iff } (\pi_L(\varphi) \subseteq \Psi) \wedge (\Psi \cap \pi_R(\varphi) = \emptyset)$$

$$\Psi \vdash a \dot{\leftrightarrow} b \text{ iff } a = b$$

where $\mathbf{T}$, $\mathbf{C}$, and $\mathbf{A}$ are nominal data types built over the nominal set E, and π_L, π_R are the standard left/right projection functions for pairs.

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.

The conditions $\mathbf{C}$ are pairs of subsets of events, which intuitively will hold the enabling conditions for an event, i.e., the left set holding those events it depends on and the right set holding those events it is in conflict with. The assertions $\mathbf{A}$ intuitively can be understood as capturing the set of all executed events, i.e., a configuration of the event structure. Channel equivalence is equality of event names, as in standard pi-calculus. Composition of two assertions is the union of the sets. The entailment $\vdash$ intuitively captures when events may fire, thus describing when events are enabled by a configuration.

Definition 3.1.2 (event structures to eventPsi) *We define a function* ESPSI *which given an event structure $\mathcal{E} = (E, \leq, \sharp)$ and a configuration C of $\mathcal{E}$, returns an eventPsi process $P_E = |_{e \in E} P_e$ with $P_e = (\!|\{e\}|\!)$ if $e \in C$, otherwise $P_e = \mathbf{case}\ \varphi_e : \overline{e}\langle e \rangle.(\!|\{e\}|\!)$, where $\varphi_e = (\leq e, \sharp e)$.*

A process generated by the ESPSI function is built up from smaller "event processes" put in parallel. These come in two forms: those corresponding to the events in the configuration of the translated event structure (i.e., those that already happened), and processes corresponding to events that have not happened yet. For the later we use a condition φ_e that contains the set $\leq e$ of events e is depending on and the set $\sharp e$ of events e is in conflict with. Together these two sets along with the frame of the entire psi-process, decide, through the entailment, if the event can execute or not. When an event happens we will have a transition over the channel with the same name as the event. Usually an event structure is encoded into eventPsi starting from the empty configuration, i.e., with no behaviour.

Example 3.1.3 *We take the event structure from Example 2.2.3 which we will now call ES, and where* ESPSI$(ES, C_{ES}) = P_{ES}$*. Let us first recall that $ES = (\{a, b, c, d\}, \{(a, c), (b, d)\}, \{(b, c), (c, b), (d, a), (a, d), (c, d), (d, c)\}, l)$, and we assume that $C_{ES} = \emptyset$. We have that $\leq_a = \emptyset, \sharp_a = \{d\}, \leq_b = \emptyset, \sharp_b = \{c\}, \leq_c = \{a\}, \sharp_c = \{b, d\}, \leq_d = \{b\}, \sharp_d = \{a, c\}$ from the relations, and thus can create $\varphi_a = (\leq_a, \sharp_a), \varphi_b = (\leq_b, \sharp_b), \varphi_c = (\leq_c, \sharp_c), \varphi_d = (\leq_d, \sharp_d)$. Further we have that $P_{ES} = P_a | P_b | P_c | P_d$ where $P_a = \mathbf{case}\ \varphi_a : \overline{a}\langle a \rangle.(\!|\{a\}|\!)$ as $a \notin C_{ES}$, $P_b = \mathbf{case}\ \varphi_b : \overline{b}\langle b \rangle.(\!|\{b\}|\!)$ as $b \notin C_{ES}$, $P_c = \mathbf{case}\ \varphi_c : \overline{c}\langle c \rangle.(\!|\{c\}|\!)$ as $c \notin C_{ES}$ and $P_d = \mathbf{case}\ \varphi_d : \overline{d}\langle d \rangle.(\!|\{d\}|\!)$ as $d \notin C_{ES}$*

This gives us the eventPsi process P_{ES} where each event has been transformed into its sub process for psi-calculi, each one with its guarded output behind a condition. The frame of P_{ES} is here $\mathcal{F}(P_{ES}) = 1 = \emptyset$ as we have no assertions in P_{ES} other than the identity assertion which is always present. We can also look and see that as $\emptyset \vdash \varphi_a$ due to $(\emptyset \subseteq \emptyset) \wedge (\emptyset \cap \{d\} = \emptyset)$, P_a may happen, similar we have that P_b may happen as $\emptyset \vdash \varphi_b$ due to $(\emptyset \subseteq \emptyset) \wedge (\emptyset \cap \{c\} = \emptyset)$. For both P_c and P_d we do not have that the case condition is allowed by entailment, if we try entail φ_c we will end up with $\{a\} \subseteq \emptyset$ which is not true, and similar we can see that φ_d is not entailed. This leaves us with that only a and b are allowed to happen in the ESPSI(ES, C_{ES}) *process we made, this are the same events that can happen in the overlaying event structure.*

Example 3.1.4 *We take the same event structure we had in Example 3.1.3, but instead of having $C_{ES} = \emptyset$ we assume that it is $C_{ES} = \{a\}$. This would change P_a as we in Example 3.1.3 had that $a \notin C_{ES}$, but as $a \in \{a\}$ we must have that $P_a = (\!|\{a\}|\!)$ from Def 3.1.2, the event processes for the other three events will not be changed as they still not in the configuration we get as parameter. This will change the frame of the process we make to $\mathcal{F}(P_{ES}) = 1 \otimes (\!|\{a\}|\!) = \emptyset \cup \{a\} = \{a\}$, while also changing which conditions are being entailed by the assertion. We have that b is not affected by this change as $(\emptyset \subseteq \{a\}) \vee (\{a\} \cap \{c\} = \emptyset)$, and thus is still able to* *happen. For c and d we had that neither of them was allowed to happen due to their condition not being entailed. Now we have that for φ_c the part of the entailment that was preventing it from being entailed earlier is $\{a\} \subseteq \{a\}$ and this is now correct, and as $\{a\} \cap \{b, d\} = \emptyset$, we have that φ_c is enabled and c can happen. On other hand d still cannot happen as $\{b\} \subseteq \{a\}$ is not true.*

Lemma 3.1.5 (correspondence configuration–frame) *For any event structure $\mathcal{E}$ and configuration $C_\mathcal{E}$, the frame of the* eventPsi *process* $\textsc{espsi}(\mathcal{E}, C_\mathcal{E})$ *corresponds to the configuration $C_\mathcal{E}$.*

Proof: Denote $\textsc{espsi}(\mathcal{E}, C_\mathcal{E}) = P_E$ as in Def 3.1.2. The frame of P_E is the composition with $\otimes$ of the frames of P_e for $e \in E$. As P_e is either $(\!|\{e\}|\!)$ if $e \in C_\mathcal{E}$ or **case** $\varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)$ then the frame of P_e would be either $\mathcal{F}((\!|\{e\}|\!)) = \{e\}$ or $\mathcal{F}(\textbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)) = \mathbf{1}$. Thus the frame of P_E is the $\otimes$ of $\mathbf{1}$'s and all events in $C_\mathcal{E}$, thus having that the frame is the union of all events in $C_\mathcal{E}$ □

This lemma intuitively says that any time we make a new eventPsi process, we know that the configuration set is the same as the set our eventPsi process has as its assertion.

Lemma 3.1.6 (transitions preserve configurations) *For some event structure $\mathcal{E}$ and some configuration of it $C_\mathcal{E}$, any transition from this configuration $C_\mathcal{E} \xrightarrow{e} C'_\mathcal{E}$ is matched by a transition $\emptyset \rhd \textsc{espsi}(\mathcal{E}, C_\mathcal{E}) \xrightarrow{\overline{e}e} \textsc{espsi}(\mathcal{E}, C'_\mathcal{E})$ in the corresponding* eventPsi*-process.*

Proof: Before the event e is executed we have that our eventPsi-process $\textsc{espsi}(\mathcal{E}, C_\mathcal{E})$ can be written in the form $P = \textbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)\,|Q$. By Lemma 3.1.5 we know that the frame of P is the same as $C_\mathcal{E}$, i.e., we have that $\mathcal{F}(P) = \mathbf{1} \otimes \mathcal{F}(Q) = \Psi_Q = C_\mathcal{E}$ before e has happened, and $e \notin C_\mathcal{E}$.

We can observe the transition between eventPsi-processes by the following proof tree, using the transition rules of psi-calculi.

$$\cfrac{\cfrac{\cfrac{\Psi_Q \otimes \emptyset \vdash e \dot{\leftrightarrow} e}{\Psi_Q \otimes \emptyset \rhd \overline{e}\langle e\rangle.(\!|\{e\}|\!) \xrightarrow{\overline{e}e} (\!|\{e\}|\!)}\ \textsc{out} \qquad \Psi_Q \otimes \emptyset \vdash \varphi_e}{\Psi_Q \otimes \emptyset \rhd \textbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!) \xrightarrow{\overline{e}e} (\!|\{e\}|\!)}\ \textsc{Case}}{\emptyset \rhd \textbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|\{e\}|\!)\,|Q \xrightarrow{\overline{e}e} (\!|\{e\}|\!)|Q}\ \textsc{Par}$$

An event e can happen if the corresponding condition in the **case** construct is entailed by the appropriate assertion $\Psi_Q \vdash \varphi_e$. This forms the right condition of the ($\textsc{case}$) rule, saying that all the preconditions of e are met, and e is not in conflict with any event that has happened. This condition is met because $C_\mathcal{E} = \Psi_Q$ and the assumption of the lemma, i.e., the existence of the step, which implies that e is enabled by the configuration $C_\mathcal{E}$, meaning exactly what the definition of the entailment relation needs.

After $\xrightarrow{\overline{e}e}$ has happened we have $P' = (\!|\{e\}|\!)|Q$ and $\mathcal{F}(P') = \mathcal{F}((\!|\{e\}|\!)) \otimes \mathcal{F}(Q) = \{e\} \cup \Psi_Q$, meaning that the frame of P' corresponds to $C'_\mathcal{E} = C_\mathcal{E} \cup \{e\}$. From Def 3.1.2 it is easy to see that $\textsc{espsi}(\mathcal{E}, C'_\mathcal{E}) = (\!|\{e\}|\!)|Q$. □

By this we have that transitions in an event structure, and the corresponding transitions in the eventPsi-process we get from giving the event structure to $\textsc{espsi}$, will make that the configuration of the event structure and the frame of the eventPsi remains equal, for equal runs.

Example 3.1.7 *We can look at the process from Example 3.1.3 now called P_1, and the process from Example 3.1.4 now called P_2. The only difference between these two processes was that one was made before any events had happened in the overlaying event structure, and the other after event a had happened.*

We have that for an event to happen in eventPsi, that there is an output over the channel with this event's name. From Example 3.1.3 we already determined that a may happen under the frame $\mathcal{F}(P_1) = \emptyset$. This frame was the same as the configuration we had when we made P_1 using ESPSI. *A transition on* **case** $\varphi_a : \overline{a}\langle a\rangle.(\!|\{a\}|\!)$ *would result in the transition label $\overline{a}a$ and result after transition as the process $(\!|\{a\}|\!)$. We can see that we with this transition went from the P_a process we had in P_1 to the P_a process we had in P_2. This transition does not affect any of the other processes we have in P_1 and we can see that we after the transition in P_1 get the same process with the same frame as the process P_2 that we made in Example 3.1.4, where we did a transition on a in the event structure before making an eventPsi-process.*

Theorem 3.1.8 (preserving concurrency) *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 $C_\mathcal{E}$ s.t. $\emptyset \rhd \text{ESPSI}(\mathcal{E}, C_\mathcal{E}) \xrightarrow{e} P_1 \xrightarrow{e'} P_2$ and $\emptyset \rhd \text{ESPSI}(\mathcal{E}, C_\mathcal{E}) \xrightarrow{e'} P_3 \xrightarrow{e} P_4$ with $P_2 = P_4$.*

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., $\underline{\leq} e \subseteq C$ and $\underline{\leq} e' \subseteq C$, and does not contain any of the two events, i.e., $e, e' \notin C$ (cf. Remark 2.2.5). Take this configuration as the one $C_\mathcal{E}$ sought in the theorem. Therefore we have the following steps in the event structure: $C_\mathcal{E} \xrightarrow{e} C_\mathcal{E} \cup e$, $C_\mathcal{E} \xrightarrow{e'} C_\mathcal{E} \cup e'$, $C_\mathcal{E} \cup e \xrightarrow{e'} C_\mathcal{E} \cup \{e, e'\}$, and $C_\mathcal{E} \cup e' \xrightarrow{e} C_\mathcal{E} \cup \{e, e'\}$.

Since $C_\mathcal{E}$ is reachable from the root then by Lemma 3.1.6 all the steps are preserved in the behaviour of the eventPsi-process $\text{ESPSI}(\mathcal{E}, \emptyset)$, meaning that $\text{ESPSI}(\mathcal{E}, C_\mathcal{E})$ is reachable from (i.e., part of the behaviour of) $\text{ESPSI}(\mathcal{E}, \emptyset)$.

Since $e, e' \notin C_\mathcal{E}$ we have that $\text{ESPSI}(\mathcal{E}, C_\mathcal{E})$ is in the form $P_0 = P_e | P_{e'} | Q$ with P_e and $P_{e'}$ processes of kind **case**. From Lemma 3.1.5 we know that the frame of $\text{ESPSI}(\mathcal{E}, C_\mathcal{E})$ is the assertion corresponding to $C_\mathcal{E}$, which is $\mathcal{F}(P_e|P_{e'}|Q) = \{\emptyset\} \cup \{\emptyset\} \cup \Psi_Q = \Psi_Q$.

From Lemma 3.1.6 we see the transitions between the eventPsi-processes: $\emptyset \rhd \text{ESPSI}(\mathcal{E}, C_\mathcal{E}) \xrightarrow{e} P_1 \xrightarrow{e'} P_2$ with $P_2 = (\!|e|\!) \mid (\!|e'|\!) \mid Q$ as well as $\emptyset \rhd \text{ESPSI}(\mathcal{E}, C_\mathcal{E}) \xrightarrow{e'} P_3 \xrightarrow{e} P_4$ with $P_4 = (\!|e|\!) \mid (\!|e'|\!) \mid Q$. We thus have the expected interleaving diamond.

As a side, 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 say that $e \in \mathcal{F}(P_1) \wedge e' \notin \mathcal{F}(P_1) \wedge e' \in \mathcal{F}(P_3) \wedge e \notin \mathcal{F}(P_3) \wedge \mathcal{F}(P_1) \otimes \mathcal{F}(P_3) = \mathcal{F}(P_4)$. With Lemma 3.1.8 these can be correlated with configurations and the we can see the definition of concurrency with configurations from Remark 2.2.5. □

The proof of Theorem 3.1.8 hints at an opposite result, stating a true concurrency rule for eventPsi-processes. 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 3.1.9 (independence diamonds) *For any event structure $\mathcal{E}$, in the corresponding eventPsi-process* $\text{ESPSI}(\mathcal{E}, \emptyset)$*, for any interleaving diamond* $\emptyset \rhd \text{ESPSI}(\mathcal{E}, C_\mathcal{E}) \xrightarrow{e} P_1 \xrightarrow{e'} P_2$ *and* $\emptyset \rhd \text{ESPSI}(\mathcal{E}, C_\mathcal{E}) \xrightarrow{e'} P_3 \xrightarrow{e} P_4$ *with* $P_2 = P_4$*, for some configuration* $C_\mathcal{E} \in \mathcal{C}_\mathcal{E}$*, we have that the events* $e||e'$ *are concurrent in* $\mathcal{E}$*.*

Proof : Since $\text{ESPSI}(\mathcal{E}, C_\mathcal{E})$ has two outgoing transitions labelled with the events e and e' it means that $\text{ESPSI}(\mathcal{E}, C_\mathcal{E})$ is in the form $P_0 = P_e|P_{e'}|Q$ with P_e and $P_{e'}$ processes of kind **case**. From Lemma 3.1.5 we know that the frame of $\text{ESPSI}(\mathcal{E}, C_\mathcal{E})$ is the assertion corresponding to $C_\mathcal{E}$, which is $\mathcal{F}(P_e|P_{e'}|Q) = \{\emptyset\} \cup \{\emptyset\} \cup \Psi_Q = \Psi_Q$.

We thus have that $e, e' \notin \Psi_Q$ and $P_0 \xrightarrow{e} P_1$ and $P_0 \xrightarrow{e'} P_3$. This means that for these two transitions to be possible it must be that the precondition for e and e' respectably 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 $\leq e$ and $\pi_L(\varphi'_e)$ the set $\leq e'$ we have the two parts of the Definition 2.2.2 that concern $\leq$ for the casual 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 $P_1 \xrightarrow{e'} P_2$ to happen we must have that $e \notin \pi_R(\varphi_{e'})$ and for $P_3 \xrightarrow{e} P_4$ 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 2.2.2 concerning the conflict relation, i.e., $\neg(e' \sharp e)$. This completes the proof, showing $e||e'$. □

Theorem 3.1.8 and Theorem 3.1.9 make sure that we catches the independence relation that exists in event structures nicely in our psi-calculi instance, and let us see when two events are independent of each other looking at the transitions we get in the interleaving semantics of psi-calculi. Though we are able to capture the independence relation for event structures, are this more due to the fact of how nicely this independence relation work, than a strength in psi-calculi, and not certain we would be able to find such an independence relation for all other true concurrency models when instantiated in psi-calculi.

We have seen that the eventPsi-processes that we obtain from event structures in Definition 3.1.2 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 3.1.1. The question is which of all these eventPsi-processes correspond exactly to event structures? We want to have syntactic restrictions on how to write eventPsi-process terms so that we are sure that there exists an event structure corresponding to each such restricted process term.

Theorem 3.1.10 (syntactic restrictions) *Consider eventPsi-process terms built only with the following grammar:*

$$P_{ES} := (\!|e|\!) \mid \textbf{case}\ \varphi : \overline{e}\langle e\rangle.(\!|e|\!) \ \mid P_{ES} \mid P_{ES}$$

Moreover, a term P_{ES} *has to respect the following constraints, for any* $\varphi_e, \varphi_{e'}$ *from* **case** $\varphi : \overline{e}\langle e\rangle.(\!|e|\!)$ *respectively* **case** $\varphi' : \overline{e'}\langle e'\rangle.(\!|e'|\!)$ *:*

1. *conflict:* $e \notin \pi_R(\varphi_e)$ *and* $e' \in \pi_R(\varphi_e)$ *iff* $e \in \pi_R(\varphi_{e'})$*;*
2. *causality:* $e \notin \pi_L(\varphi_e)$ *and if* $e \in \pi_L(\varphi_{e'})$ *then* $e' \notin \pi_L(\varphi_e) \wedge \pi_L(\varphi_e) \subset \pi_L(\varphi_{e'})$*;*
3. *executed events:* P_{ES} *cannot have both* $(\!|e|\!)$ *and* **case** $\varphi : \overline{e}\langle e\rangle.(\!|e|\!)$ *for any* e*, nor multiples of each.*

For any such restricted process P_{ES} *there exists an event structure* $\mathcal{E}$ *and configuration* $C_\mathcal{E} \in \mathcal{C}_\mathcal{E}$ *s.t.*

$$\text{ESPSI}(\mathcal{E}, C_\mathcal{E}) = P_{ES}.$$

Proof: From a eventPsi-process P_{ES} defined as in the statement of the theorem, we show how to construct an event structure $\mathcal{E} = (E, \leq, \sharp)$ and a configuration $C_{\mathcal{E}}$. We have that P_{ES} is built up of assertion processes and case guarded outputs, i.e., $P_{ES} = (\,|_{e \in E_c} (\!|e|\!)) \;|\; (\,|_{f \in E_r} \mathbf{case}\ \varphi_f : \overline{f}\langle f\rangle.(\!|f|\!)\,)$.

Because of the third restriction on P_{ES} we know that E_c and E_r are sets, as no multiples of the same process can exist. Moreover, these two sets are disjoint. For otherwise, assume we have $(\!|e|\!)|\mathbf{case}\ \varphi_e : \overline{e}\langle e\rangle.(\!|e|\!)$ part of P_{ES}. This is the same as if e has happened already and e may happen in future, which cannot be the case for event structures.

We take $C_{\mathcal{E}}$ 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: $\leq = \cup_{e \in E_r}\{(e', e) | e' \in \pi_L(\varphi_e)\}$ 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 second restriction on P_{ES}. Transitivity is easy to obtain from 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 $\text{ESPSI}(\mathcal{E}, C_{\mathcal{E}}) = P_{ES}$. The encoding function ESPSI takes all events from $C_{\mathcal{E}}$ to the left part of the P_{ES}, whereas the remaining events, i.e., from E_r are taken to **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. □

We can see that for requirement one, we must have that an event cannot be in conflict with itself, but that all events this event has in its set of conflicting events, this event must be in their sets of conflicting events.

Requirement two says that an event cannot be a condition for itself. If an event is a condition for another event, then all of the events this first event has as conditions, are conditions for the second event.

Requirement three gives us that no event should be able to happen more than once, and cannot be both executed and not executed same time. So only one sub-process is allowed for each event, in either the style of an single assertion or as an output process reducing to an assertion.

Remark 3.1.11 *In Event Structures the E may be infinite. The configurations though are always finite. So we may generate infinite terms in Def 3.1.2. Infinite terms are not desired, for practical reasons. But there are works with infinite terms, like infinite summation. Recursion of pi-calculi and replication of psi-calculus are the way to obtain infinite parallel components, as we generate from event structures.*

In our case we also wanted to have the nice presentation, therefore we opted to generate infinite terms. From these terms it is clear and natural to see the correlation with the event structures. We work the same as in event structures, by tacitly having infinitely many events, thus infinite parallel processes.

3.2 Refinement

In the previous section we have seen results that give close correlations between event structures and eventPsi-processes. In Section 2.3 we described the important method of action refinement for event structures. Here we would be interesting in having a similar method of refinement for the eventPsi-processes and see how it can be correlated to the corresponding event structures.

Definition 3.2.1 *Given a refinement function for event structures ref, we define a function ref^P that refines an eventPsi-process to a new one over the names*

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

A process P with frame $\mathcal{F}(P) = \Psi_P$ is refined into a process

$$ref^P(P) = |_{(e,e') \in T^P} P_{(e,e')}$$

with

$$P_{(e,e')} = (\!|\{(e, e')\}|\!), \textit{if } e \in \Psi_P$$

otherwise

$$P_{(e,e')} = \textbf{case } \varphi_{(e,e')} : \overline{(e, e')}(e, e').(\!|\{(e, e')\}|\!)$$

with the conditions being

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

where

$$\leq(e, e') = \{(d, d') \mid d \in \pi_L(\varphi_e) \vee (d = e \wedge d' \in \leq_{ref(l(d))} e)\}$$

and

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

The new names are pairs of a parent event name (i.e., from the original process) and one of the event names from the refinement processes. We do not end up outside the eventPsi instance because we can rename any pair by names from E. 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; rename any pair by an event from E while preserving the order, thus making T^P the same as the $\mathbf{T}$ of eventPsi.

We make new conditions for each of the new names (e, e'), where $\leq(e, e')$ contains all pairs of names s.t. either the left part is a condition for e, or the left part is the same as e but the right part is a condition for e'. The conflicts set $\sharp(e, e')$ contains all pairs of names with the first part a conflict for e. The refinement generates for each new pair one processes 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 or not.

Example 3.2.2 *We look on the eventPsi process P we made in Example 3.1.3, and the refinement function ref used in Example 2.3.3. We have that only one event a are to be refined into something other than itself, and this is into $(\{e, f\}, \{(e, f)\}, \emptyset, l_{ref(a)})$. We will follow the way we wrote the new names for events b, c and d that we used in Example 2.3.3. From ref we can make ref^P that will take our P and return a refined version of it P_{ref}, similar to when we used* *ref in Example 2.3.3. In this ref^P we get that we change nothing in φ_b as a was not part of any of its two sets. For φ_c we get that $\leq_c$ gets changed to $\leq c = \{(a,e),(a,f)\}$ as $a \in \leq_c$ in P, but the conflict set does not change as $a \notin \sharp_c$ in P. For φ_d we have that while $\leq_d$ remains the same $\sharp_d$ gets refined into $\{(a,e),(a,f),c\}$ due to $a \in \sharp_d$ in P.*

For P_a we have that it gets refined into the two processes $P_{(a,e)}$ and $P_{(a,f)}$. The condition $\varphi_{(a,e)}$ is made up of the two sets $\leq_{(a,e)}$ and $\sharp_{(a,e)}$, where $\leq_{(a,e)} = \emptyset$ as neither a nor e have any conditions to keep, and $\sharp_{(a,e)}$ is $\{d\}$ from a being in conflict with d in P_a. For the condition $\varphi_{(a,f)}$ we have that $\sharp_{(a,f)}$ is $\{d\}$ from a being in conflict with d, while we have that $\leq_{(a,f)} = \{(a,e)\}$ as $e \leq f$.

From these conditions we can move ahead and create the processes:
$P_{(a,e)} = \textbf{case}\ \varphi_{(a,e)} : \overline{(a,e)}\langle(a,e)\rangle.(\!|\{(a,e)\}|\!)$,
$P_{(a,f)} = \textbf{case}\ \varphi_{(a,f)} : \overline{(a,f)}\langle(a,f)\rangle.(\!|\{(a,f)\}|\!)$
that are the new sub-processes we get in place for P_a. After we change the conditions needed for the other three event-processes we get that $P_{ref} = P_{(a,e)}|P_{(a,f)}|P_b|P_c|P_d$.

Theorem 3.2.3 (refinement of eventPsi corresponds to refinement in ES) *For a prime event structure $\mathcal{E}$ we have that:* $\quad \text{ESPSI}(ref(\mathcal{E}),\emptyset) = ref^P(\text{ESPSI}(\mathcal{E},\emptyset))$.

Proof: As $\mathbf{T} = E$ and as T^P is built from $\mathbf{T}$ with the same rules as E_{ref} is built from E we have that $T^P = E_{ref}$. 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 fount 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 neither sides.

The **case** processes on the left side are those generated by ESPSI from the pairs events returned by the *ref* from the event structure. This means that for each pair we have its condition built up as in the Definition 2.3.2. On the right side we have **case** processes for the original process before the refinement, with their respective conditions. But the ref^P replaces these with many **case** processes, one for each new pair, and for each new **case** process, the conditions are build exactly as the *ref* is defining them. This says that we have the same number of **case** processes on both sides of the equality, and they have the same conditions. □

Example 3.2.4 *In Example 3.2.2 we used a refinement function ref^P to refine our eventPsi-process from Example 3.1.3, based on the ref function from Example 2.3.3 and made the process P_{ref}. This time we will take the event structure we got after refining in Example 2.3.3, and convert it into an eventPsi-process using the ESPSI function. To recall we had that the event structure after refining was*
$ref(ES) = (\{(a,e),(a,f),b,c,d\},\{((a,e),c),((a,f),c),(b,d)\},$
$\{((a,e),d),((a,f),d),(d(a,e)),(d,(a,f)),(d,c),(c,d),(b,c),(c,b)\})$.

*Having five events and no configuration we need to make five **case** processes, and their conditions respectively. These conditions will become $\varphi_{(a,e)} = (\emptyset,\{c\}), \varphi_{(a,f)} = (\{(a,e)\},\{c\}), \varphi_b = (\emptyset,\{c\}), \varphi_c = (\{(a,e),(a,f)\},\{b,d\}), \varphi_d = (\{b\},\{(a,e),(a,f),c\})$. We can now look and see that these conditions are the same conditions that we got after refining the process P in Example 3.2.2.*

Each of the sub-processes are made in the same way as we make the sub-process for the event (a, e) *here:* $P_{(a,e)} = \textbf{case}\ \varphi_{(a,e)} : \overline{(a, e)}\langle(a, e)\rangle.(\!|\{(a, e)\}|\!)$. *When placed together in parallel composition we get that the* **eventPsi**-*process we make is* $P_{ref(ES)} = P_{(a,e)}|P_{(a,f)}|P_b|P_c|P_d$. *It is easy to see that this process, are the same as the process we made in Example 3.2.2.*

Chapter 4

DCR-graphs as psi-calculi

We achieved a rather natural and intuitive translation of the prime event structures into an instance of psi-calculi. 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. Noteworthy is that we have not used the communication mechanism of psi-calculus, which is known to increase expressiveness.

We try to extend this approach from event structures to the DCR-graphs. But it appears that we need the communication constructs on processes to keep track of the current marking of a DCR-graph. The particularities and expressiveness of DCR-graphs do not allow for a simple way of updating the marking, as was the case for event structures when just union with the newly executed event was enough. But we obtain a nice encoding once we use the communication, outputting a term representing the current marking, and incorporating an idea of generation (or age) of an assertion, where assertion composition keeps the newest generation. We thus have a way to just use the newest assertion for entailments, and we get a more natural encoding for DCR-graphs in a psi-calculus instance. We can then see correlations with the previous encoding of the event structures. The markings are kept in the assertions, i.e., as the frame of the process, the same as we did with the configurations of the event structures. Case processes are used for each event of the DCR-graph, and the conditions of the case processes capture the conditions that the events of a DCR-graph depend on to be enabled in a marking. The entailment relation then captures the enabling of events by markings.

Definition 4.0.5 (dcrPsi instance) *We define an instantiation of Psi-calculi called dcrPsi by providing the following definitions:*

$$\mathbf{T} \stackrel{def}{=} \{m\} \cup \mathbf{A}$$

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

where E is a nominal set and $\mathbb{N}$ is the nominal data structure capturing natural numbers using a successor function $s(\cdot)$ and generator 0, whereas m is a single name used for communication;

$$\mathbf{C} \stackrel{def}{=} 2^E \times 2^E \times E$$

$$\dot{\leftrightarrow} \stackrel{def}{=} =$$

$$\mathbf{1} \stackrel{def}{=} (\emptyset, \emptyset, \emptyset, 0)$$

$$\llparenthesis(Ex, Re, In, G)\rrparenthesis \otimes \llparenthesis(Ex', Re', In', G')\rrparenthesis \overset{def}{=} \begin{cases} \llparenthesis(Ex, Re, In, G)\rrparenthesis & \textit{if}\, G > G' \\ \llparenthesis(Ex', Re', In', G')\rrparenthesis & \textit{if}\, G < G' \\ \llparenthesis(Ex \cup Ex', Re \cup Re', In \cup In', G)\rrparenthesis & \textit{if}\, G = G' \end{cases}$$

where the comparison $G < G'$ is done using sub-term relation, eg., $s(N) > N$. Entailment $\vdash$ is defined as:

$$\llparenthesis(Ex, Re, In, G)\rrparenthesis \vdash (Co, Mi, e) \quad \textit{iff} \quad e \in In \wedge (In \cap Co) \subseteq Ex \wedge ((In \cap Mi) \cap Re) = \emptyset.$$

$$\llparenthesis(Ex, Re, In, G)\rrparenthesis \vdash a \leftrightarrow b \text{ iff } a = b$$

Terms can be either a name m, which we will use for communications, or assertions which will be the data communicated. Assertions are a tuple of three sets of events, and a natural number we intend to hold the *generation* of the assertion. The first set is meant to capture what events have been executed, the second set for those events that are pending responses, and the third set for those events that are included. These three sets mimic the same sets that the marking of a DCR-graph contains. The generation number is used 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.[1] This makes the composition associative, commutative, compositional, and with identity defined to be the tuple with empty sets and lowest possible generation number.

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.

The entailment definition 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). Compare the example below with the definition of enabling from DCR-graphs

$$\llparenthesis(Ex, Re, In, G)\rrparenthesis \vdash (\rightarrow\!\bullet e, \rightarrow\!\diamond e, e) \quad \text{iff} \quad e \in Ex \wedge (In \cap \rightarrow\!\bullet e) \subseteq Ex \wedge ((In \cap \rightarrow\!\diamond e) \cap Re) = \emptyset.$$

Definition 4.0.6 *We define the function* DCRPSI *which takes a DCR-graph $(E, M \rightarrow\!\bullet, \bullet\!\rightarrow, \rightarrow\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$ with a distinguished marking $M = (Ex', Re', In')$ and returns a* **dcrPsi** *process*

$$P_{dcr} = P_s \,|\, P_E$$

where

$$P_s = \llparenthesis(Ex', Re', In', 0)\rrparenthesis \,|\, \overline{m}\langle(Ex', Re', In', 0)\rangle.\mathbf{0}$$

and

$$P_E = \,|_{\,e \in E} P_e$$

with

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

[1]For technical reasons, when we compose two assertions with the same generation number we obtain an assertion where the sets are the union between the associated sets in each assertion, and the generation number is unchanged.

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

The process P_{dcr} generated by DCRPSI contains a starting processes P_s that models the initial marking of the encoded DCR-graph as an assertion process, and also communicates this assertion (i.e., the current marking) on the channel m. The rest of the process, i.e., P_E captures the actual DCR-graph, being a parallel composition of processes P_e for each of the events of the encoded DCR-graph. The events in a DCR-graph can happen multiple times, hence the use of the replication operation as the outermost operator. Each event is encoded, following the ideas for event structures, using the **case** construct with a single guard φ_e. The guard contains the conditions for the event e that need to be checked against the current marking (i.e., the assertion) to decide if the event is enabled; these conditions are the set of events that are prerequisites for e (i.e., $\rightarrow\!\!\bullet e$) and the set of milestones related to e. 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_s process, but with an updated marking. The updating of the marking follows the same definition as in the DCR-graphs.

Example 4.0.7 *We take the DCR-graph we formalized in Example 2.5.3, where we have the DCR-graph* $G = (E, M, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%, L, l)$ *where*

$E = \{go, co, tp, fa, rd\}$,
$M = (\emptyset, \emptyset, \{go, co, tp\})$
$\rightarrow\!\!\bullet = \{(go, co), (co, tp)\}$,
$\bullet\!\!\rightarrow = \{(go, co), (co, tp), (tp, fa), (tp, rd), (rd, co)\}$,
$\rightarrow\!\!\diamond = \{(co, tp)\}$,
$\rightarrow\!\!+ = \{(tp, fa), (tp, rd), (rd, co), (rd, tp)\}$,
$\rightarrow\!\!\% = \{(go, go), (co, co), (tp, tp), (rd, rd), (fa, fa), (fa, rd), (rd, fa)\}$.

From this DCR-graph we make the dcrPsi *process*

$P_{dcr} = \text{DCRPSI}(G) = P_s | P_{go} | P_{co} | P_{tp} | P_{rd} | P_{fa}$

We make our start process P_s *which is the part where our initial assertion and our initial output process is to be:*

$$(\!|(\emptyset, \emptyset, \{go, co, tp\}, 0)|\!) \,|\, \overline{m}\langle(\emptyset, \emptyset, \{go, co, tp\}, 0)\rangle.\mathbf{0}$$

For the five other processes P_{go}, P_{co}, P_{tp}, P_{rd} *and* P_{fa}. *We have that they are all made using the* P_e *template from Def 4.0.6, causing them to be replicating processes:*

$P_{go} = !(\mathbf{case}\ \varphi_{go} : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.$
$(\overline{m}\langle(X_E \cup \{go\}, (X_R \setminus \{go\}) \cup go\bullet\!\!\rightarrow, (X_I \setminus go \rightarrow\!\!\%) \cup go \rightarrow\!\!+, s(X_G))\rangle.\mathbf{0}\ |$
$(\!|(X_E \cup \{go\}, (X_R \setminus \{go\}) \cup go\bullet\!\!\rightarrow, (X_I \setminus go \rightarrow\!\!\%) \cup go \rightarrow\!\!+, s(X_G))|\!)\,)\,)$

$P_{co} = !(\mathbf{case}\ \varphi_{co} : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.$
$(\overline{m}\langle(X_E \cup \{co\}, (X_R \setminus \{co\}) \cup co\bullet\!\!\rightarrow, (X_I \setminus co \rightarrow\!\!\%) \cup co \rightarrow\!\!+, s(X_G))\rangle.\mathbf{0}\ |$
$(\!|(X_E \cup \{co\}, (X_R \setminus \{co\}) \cup co\bullet\!\!\rightarrow, (X_I \setminus co \rightarrow\!\!\%) \cup co \rightarrow\!\!+, s(X_G))|\!)\,)\,)$

$P_{tp} =!(\mathbf{case}\ \varphi_{tp} : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.$
$(\overline{m}\langle(X_E \cup \{tp\}, (X_R \setminus \{tp\}) \cup co \bullet\!\!\rightarrow, (X_I \setminus tp \rightarrow\!\!\%) \cup tp \rightarrow\!\!+, s(X_G))\rangle.\mathbf{0} \mid$
$(\!|(X_E \cup \{tp\}, (X_R \setminus \{tp\}) \cup tp \bullet\!\!\rightarrow, (X_I \setminus tp \rightarrow\!\!\%) \cup tp \rightarrow\!\!+, s(X_G))|\!))\ )$
$P_{fa} =!(\mathbf{case}\ \varphi_{fa} : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.$
$(\overline{m}\langle(X_E \cup \{fa\}, (X_R \setminus \{fa\}) \cup fa \bullet\!\!\rightarrow, (X_I \setminus fa \rightarrow\!\!\%) \cup fa \rightarrow\!\!+, s(X_G))\rangle.\mathbf{0} \mid$
$(\!|(X_E \cup \{fa\}, (X_R \setminus \{fa\}) \cup fa \bullet\!\!\rightarrow, (X_I \setminus fa \rightarrow\!\!\%) \cup fa \rightarrow\!\!+, s(X_G))|\!))\ )$
$P_{rd} =!(\mathbf{case}\ \varphi_{rd} : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.$
$(\overline{m}\langle(X_E \cup \{rd\}, (X_R \setminus \{rd\}) \cup rd \bullet\!\!\rightarrow, (X_I \setminus rd \rightarrow\!\!\%) \cup rd \rightarrow\!\!+, s(X_G))\rangle.\mathbf{0} \mid$
$(\!|(X_E \cup \{rd\}, (X_R \setminus \{rd\}) \cup rd \bullet\!\!\rightarrow, (X_I \setminus rd \rightarrow\!\!\%) \cup rd \rightarrow\!\!+, s(X_G))|\!))\ ) \mid$

Here we have that the conditions for each of these processes are:
$\varphi_{go} = (\rightarrow\!\!\bullet go, \rightarrow\!\!\diamond go, go), \varphi_{co} = (\rightarrow\!\!\bullet co, \rightarrow\!\!\diamond co, co),$
$\varphi_{tp} = (\rightarrow\!\!\bullet tp, \rightarrow\!\!\diamond tp, tp), \varphi_{af} = (\rightarrow\!\!\bullet af, \rightarrow\!\!\diamond af, af), \varphi_{rd} = (\rightarrow\!\!\bullet rd, \rightarrow\!\!\diamond rd, rd)$

Lemma 4.0.8 *For any DCR-graph $\mathcal{D}$, the frame of the corresponding process* DCRPSI$(\mathcal{D})$ *corresponds to the marking of the encoded DCR-graph (i.e., the first three components).*

Proof: DCRPSI$(\mathcal{D})$ return a dcrPsi process with only one assertion which thus is the frame. This assertion is made directly from the marking of $\mathcal{D}$ and added generation 0. □

Lemma 4.0.9 *For any DCR-graph $\mathcal{D}$, in the execution graph of the corresponding process* DCRPSI$(\mathcal{D})$ *at any execution point there will be only one output process.*

Proof: Initially we have only one output in the P_s part of DCRPSI$(\mathcal{D})$. Inductively we assume a reachable process P with only one output process. If we have any enabled input processes only one of these processes will join a communication with the single output process. All input processes are of the form P_e, which reduces with psi-calculi rules for replication and input to

$$P_e|(\overline{m}\langle(X_E \cup \{e\}, (X_R \setminus \{e\}) \cup\ \bullet\!\!\rightarrow e, (X_I \setminus \rightarrow\!\!\% e) \cup \rightarrow\!\!+ e, s(X_G))\rangle.\mathbf{0} \mid$$

$$(\!|(X_E \cup \{e\}, (X_R \setminus \{e\}) \cup\ \bullet\!\!\rightarrow e, (X_I \setminus e \rightarrow\!\!\%) \cup \rightarrow\!\!+ e, s(X_G))|\!))$$

with X_E, X_R, X_I, X_G substituted with the terms that were sent. The output process reduces to 0. We have added as many new output processes as we have removed, and as we initially only have one output process by induction we always will have only one. □

Lemma 4.0.10 *For any DCR-graph $\mathcal{D}$, in the corresponding process* DCRPSI$(\mathcal{D})$ *the message being sent will always be the same as the frame of the dcrPsi process.*

Proof: Initially, the first message being sent by P_s is by construction the same as the initial frame. The proof of Lemma 4.0.9 shows that with each communication a new assertion is added and a new sender replaces the old one. The two new terms (i.e., the assertion process and the message) are identical and have the generation part increased by one. Since the composition of assertions keeps only the assertion with the higher generation, all older assertion processes that are still present are being ignored when computing the frame of the new process. We thus have our result. □

Lemma 4.0.11 (generations count transitions) *The generation part of the frame is the same as the number of transitions we have done from the initial process.*

Proof: We use induction and assume we have done n transitions and the generation part of our frame is n' where $n = n'$. From Lemma 4.0.10 we have that the frame and message are equal, so we will be sending n as generation part of the message. After the communication a new assertion with generation $s(n')$ is added, which by the definition of assertion composition will be the new frame. By our assumption $s(n') = s(n) = n + 1$. From Lemma 4.0.8 we have that $n = n' = 0$ for the initial process, and by induction we have that this holds for any number of transitions. □

Theorem 4.0.12 (preserving transitions) *In a DCR-graph $\mathcal{D}$, for any transition $(\mathcal{D}, M) \xrightarrow{e} (\mathcal{D}, M')$ there exists a reduction between the corresponding* dcrPsi *processes* $\textsc{dcrpsi}(\mathcal{D}, M) \xrightarrow{\tau} \textsc{dcrpsi}(\mathcal{D}, M')$.

Proof: From Lemma 4.0.8 we know that the frame and marking are the same. This means that since $M \vdash e$, the corresponding condition in the $\textsc{dcrpsi}(\mathcal{D}, M)$ will be entailed by the frame. Therefore a communication is possible, i.e., a transition labelled by τ. For $M = (Ex, Re, In)$ it means that the frame of $\textsc{dcrpsi}(\mathcal{D}, M)$ is (Ex, Re, In, G). From Lemma 4.0.10 we know that the frame is always the same as the message being sent. When the transition corresponding to the event e happens the new frame of the dcrPsi becomes

$$\llparenthesis (Ex \cup \{e\}, (Re \setminus \{e\}) \cup \bullet\!\!\to e, (In \setminus e \to\!\!\%) \cup e \to\!\!+, s(G)) \rrparenthesis$$

after alpha-conversion. For a transition in a DCR-graph over the event e we get the new marking

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

which is the same as the new frame, with the exception of the generation part. □

Example 4.0.13 *We return to our process P_{dcr} from Example 4.0.7 where for ease of reading we will denote it $P_{dcr} = P_s \,|\, P_{go} \,|\, Q$ with $Q = P_{co} \,|\, P_{tp} \,|\, P_{rd} \,|\, P_{af}$. We know from Example 2.5.3 that only event go can happen initially, and thus is the only input process that is enabled. The output process is always enabled as it is never under a condition. This means that only these two processes may have transitions reducing them at the start of our run.*

We know that $go \bullet\!\!\to = \{co\}, go \to\!\!+ = \emptyset, go \to\!\!\% = \{go\}$ and thus P_{dcr} can be written as
$P_{dcr} = Q \,|\, \llparenthesis(\emptyset, \emptyset, \{go, co, tp\}, 0)\rrparenthesis \,|\, \overline{m}\langle(\emptyset, \emptyset, \{go, co, tp\}, 0)\rangle.\mathbf{0} \,|$
$!(\mathbf{case}\ \varphi_{go} : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.$
$(\overline{m}\langle(X_E \cup \{go\}, (X_R \setminus \{go\}) \cup \{co\}, (X_I \setminus \{go\}) \cup \emptyset, s(X_G))\rangle.\mathbf{0} \,|$
$\llparenthesis(X_E \cup \{go\}, (X_R \setminus \{go\}) \cup \{co\}, (X_I \setminus \{go\}) \cup \emptyset, s(X_G))\rrparenthesis)\ )$

A communication over m would cause the transition $P_{dcr} \xrightarrow{\tau} P'_{dcr}$
$P'_{dcr} = Q \,|\, \mathbf{0} \,|\, \llparenthesis(\emptyset, \emptyset, \{go, co, tp\}, 0)\rrparenthesis \,|\, \llparenthesis(\emptyset \cup \{go\}, (\emptyset \setminus \{go\}) \cup \{co\}, (\{go, co, tp\} \setminus \{go\}) \cup \emptyset, s(0))\rrparenthesis$
$|\, \overline{m}\langle(\emptyset, \emptyset, \{go, co, tp\}, 0)(\emptyset \cup \{go\}, (\emptyset \setminus \{go\}) \cup \{co\}, (\{go, co, tp\} \setminus \{go\}) \cup \emptyset, s(0))\rangle.\mathbf{0} \,|$
$!(\mathbf{case}\ \varphi_{go} : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.$

$(\overline{m}\langle(X_E \cup \{go\}, (X_R \setminus \{go\}) \cup \{co\}, (X_I \setminus \{go\}) \cup \emptyset, s(X_G))\rangle.\mathbf{0} \mid$
$(\!|(X_E \cup \{go\}, (X_R \setminus \{go\}) \cup \{co\}, (X_I \setminus \{go\}) \cup \emptyset, s(X_G))|\!)\,)\,)$

More compressed written as

$P' = Q \mid \mathbf{0} \mid (\!|(\emptyset, \emptyset, \{go, co, tp\}, 0)|\!) \mid (\!|(\{go\}, \{co\}, \{co, tp\}, s(0))|\!)$
$\mid \overline{m}\langle(\{go\}, \{co\}, \{co, tp\}, s(0))\rangle.\mathbf{0} \mid$
$!(\mathbf{case}\ \varphi_{go} : \underline{m}\langle(X_E, X_R, X_I, X_G)\rangle.$
$(\overline{m}\langle(X_E \cup \{go\}, (X_R \setminus \{go\}) \cup \{co\}, (X_I \setminus \{go\}) \cup \emptyset, s(X_G))\rangle.\mathbf{0} \mid$
$(\!|(X_E \cup \{go\}, (X_R \setminus \{go\}) \cup \{co\}, (X_I \setminus \{go\}) \cup \emptyset, s(X_G))|\!)\,)\,)$

We can see that

$(\!|(\{go\}, \{co\}, \{co, tp\}, s(0))|\!) \otimes (\!|(\emptyset, \emptyset, \{go, co, tp\}, 0)|\!) = (\!|(\{go\}, \{co\}, \{co, tp\}, s(0))|\!)$.
If we look at the three sets in this tuple we see that these are the same sets that make up the marking M' that we got in Example 2.5.3 after transition on go.

*If we look at the **dcrPsi** we get from* DCRPSI(G')*, the only difference would be in the P_s part, where we would have to use M' instead of M to generate it. This would give us*

$P_s = (\!|(\emptyset, \emptyset, \{go, co, tp\}, 0)|\!) \mid \overline{m}\langle(\{go\}, \{co\}, \{co, tp\}, 0)\rangle.\mathbf{0}$
With the exception of the generation number this is the exact same as the P_s part of P'.

Chapter 5

True concurrency semantics for Psi-calculi

5.1 Introduction

This is ongoing work where we are interested in developing a non-interleaving semantic for psi-calculi.

The semantics of psi-calculi is given through structural operational rules and adopts an interleaving approach to concurrency, in the usual style of process algebras. On the other hand, event-based models of concurrency take a non-interleaving view. Many times these form domains and are used to give denotational semantics, as e.g., done by Winskel in [Win82, WN95]. Many times non-interleaving models of concurrency can actually distinguish between interleaving and, so called, "true concurrency", as is the case with higher dimensional automata [Pra91, Pra00, vG06], configuration structures [vGP09], or Chu spaces [Gup94, Pra95].

We are interested in non-interleaving semantics for psi-calculi, and this chapter will report on our results in this direction. In particular, we are interested in a true concurrency semantics that is more operational than denotational. In other words, we would like the concurrency model that is obtained from the semantics of psi-calculus to be less like event structures or configurations structures, and more like higher dimensional automata or asynchronous transition systems [Bed88, Shi85]. (Note that Chu spaces would somewhat capture both worlds, as Pratt argues in [Pra02].)

The work of Mukund and Nielsen [MN92] is particularly attractive because it gives a non-interleaving semantics in an operational style to a process calculus, Milner's CCS [Mil80]. Usually non-interleaving models of concurrency, like Mazurkiewicz traces [Maz88], event structures [NPW79, Win88], or geometric models [GM12] s.a. higher dimensional automata [Pra91, vG06], are used in a denotational style. Domains for such models are well investigated and operations on them are used to give semantics to process calculi, e.g., Winskel for CCS [Win82], Goubault for HDAs [Gou93]. On the other hand, operational style of giving semantics using structural operational transition rules [Plo81, AFV01] for process languages obtains an interleaving concurrency semantics. Operational semantics are more intuitive and easy to understand, whereas non-interleaving models of concurrency are more expressive and adequate for capturing concurrent systems.

The achievement of Mukund and Nielsen [MN92] combines the above two aspects by going to asynchronous transition systems (ATS). These are a non-interleaving concurrency model that generalizes transition systems (i.e., the model of the SOS interleaving approach) and that has the same expressive power as the event structures [WN95, Sec.10]. Their work looks at the CCS process calculus which has only synchronization. We are interested in the psi-calculus which also has communication mechanisms. These pose extra challenges. Non-interleaving semantics in the denotational style have been recently investigated for pi-calculus by Crafa, Varacca, and Yoshida [CVY07, CVY12] using event structures as the underlying concurrency model and following the categorical techniques of Winskel and Nielsen [WN95]. We would like to extend the work on CCS using ATS, taking insights from the work using event structures, to develop a non-interleaving operational semantics for psi-calculus based on ATS.

A first step in our investigation was to extend the work of Mukund and Nielsen on CCS to the psi-calculi [BJPV11] where the work of Sangiorgi [San96] is closely related. But Sangiorgi has a different goal in his work and is based on the original dynamic approach to locations introduced in [BCHK94, BCHK93], and work on pi-calulus [Mil99]. On the other hand, Mukund and Nielsen use the approach of Aceto [Ace94] with static locations, obtaining a more clean presentation and semantics for CCS, in our opinion. It is not easy to use the work of [San96] for our semantics purposes, but the recent observations that Crafa, Varacca, and Yoshida [CVY07, CVY12] make for their denotational approach is much useful for us, as we detail further. Therefore, we have reworked the approach with static locations of Mukund and Nielsen to the setting of psi-calculus, thus treating also the extra logical aspects that psi-calculus has, compared to pi-calculus.

5.2 Background

5.2.1 Asynchronous transition systems

Asynchronous transition systems were independently introduced by Bednarczyk [Bed88] and Shields [Shi85]. The main idea is to extend transition systems by, in addition, specifying which transitions are independent of which. More accurately transitions are to be thought of as occurrences of events which bear a relation of independence. In this thesis we adopt the definition from [WN95].

Definition 5.2.1 (Asynchronous transition systems) *An synchronous transition system is a structure $ATS = (S, i, E, I, Tran)$ such that*

- $(S, i, E, Tran)$ *is a transition system*
- $I \subseteq E \times E$ *is an irreflexive and symmetric relation, called* "independence"*, satisfying the following conditions:*

  1. $e \in E \Rightarrow \exists s, s' \in S.(s, e, s') \in Tran$
  2. $(s, e, s') \in Tran$ *and* $(s, e, s'') \in Tran \Rightarrow s' = s''$
  3. $e_1 I e_2$ *and* $(s, e_1, s_1) \in Tran$ *and* $(s, e_2, s_2) \in Tran \Rightarrow$
     $\exists u : (s_1, e_2, u) \in Tran$ *and* $(s_2, e_1, u) \in Tran$
  4. $e_1 I e_2$ *and* $(s, e_1, s_1) \in Tran$ *and* $(s_1, e_2, u) \in Tran \Rightarrow$
     $\exists s_2.(s, e_2, s_2) \in Tran$ *and* $(s_2, e_1, u) \in Tran$

Condition (1) stipulates that every event in E must appear as the label of some transition in the system. The second condition guarantees that the system is deterministic. The third and fourth conditions express properties of independence: condition (3) says that if two independent events are enabled at a state, then they should be able to occur "together" and reach a common state; condition (4) says that if two independent events occur immediately after one another in the system, it should be possible for them to occur with their order interchanged.

5.2.2 CCS, locations and asynchronous transition systems

Mukund and Nielsen present in [MN92] semantics for CCS as a non-interleaving transition system. The main criteria was to develop semantics as close as possible to the original (interleaving) semantic, while also identify the concurrency present in the system in a natural way, and be simple.

This has been done by adding locations to the semantics, thus positioning where the transitions happen in the system. Through these locations one can see when two transitions are independent.

Start by fixing a set of actions $Act = \Lambda \cup \overline{\Lambda}$, where Λ is a set of names ranged over by $\alpha, \beta, ...$ and $\overline{\Lambda}$ is the corresponding set of co-names $\{\overline{\alpha} | \alpha \in \Lambda\}$. The co-name operation is considered as standard bijection such as $\overline{\overline{\alpha}} = \alpha$ for all $\alpha \in \Lambda$. The symbol $\tau \notin Act$ denotes the invisible action. $a, b, c...$ range over Act and $\mu, \nu...$ range over $Act_\tau = Act \cup \{\tau\}$. Consider also a set V of process variables and let $x, y, ...$ range over V.

The operational semantics developed in [MN92] is

$$\frac{P = \Sigma_{i \in I} \mu_i P_i \xrightarrow[{[P][P_i]}]{\mu_i} P_i}{P \xrightarrow[{[P][P_i]}]{\mu_i} P_i} \text{ (SUM)}$$

$$\frac{P_0 \xrightarrow[u]{\mu} P_0'}{P_0 || P_1 \xrightarrow[0u]{\mu} P_0' || P_1} \text{ (PAR1)}$$

$$\frac{P_0 \xrightarrow[u]{\mu} P_0'}{P_1 || P_0 \xrightarrow[0u]{\mu} P_1 || P_0'} \text{ (PAR1)}$$

$$\frac{P_0 \xrightarrow[u]{a} P_0' \qquad P_1 \xrightarrow[v]{\overline{a}} P_1'}{P_0 || P_1 \xrightarrow[\langle 0u, 1v \rangle]{\tau} P_0' || P_1'} \text{ (COM)}$$

$$\frac{P \xrightarrow[u]{\mu} P'}{P \backslash \alpha \xrightarrow[\alpha u]{\mu} P' \backslash \alpha} \text{ (RES)}$$

$$\frac{P \xrightarrow[u]{\mu} P' \qquad P \equiv P_1 \qquad P' \equiv P_1'}{P_1 \xrightarrow[u]{\mu} P_1'} \text{ (STRUCT)}$$

For a basic action preformed by a process $\Sigma_{i \in I} \mu_i P_i$, the transition is tagged with the source and target process expressions. The tag is extended with $0s$ and $1s$ on the left as the transitions are lifted through left and right branches of a parallel composition. With each communication they keep track of the tags corresponding to the two components participating in the communication. By extending the tag to the left with α, they keep track of the nesting of restrictions with respect to the overall structure of a process. This is crucial in order to determine whether or not a communication is possible even though the visible actions which make up the communications are restricted away. Finally, the rule (STRUCT) ensures that processes from the same equivalence class are capable of making exactly the same moves.

The string of $0s$ and $1s$ which is used to tag a transition when one moves up the left and right branches of a parallel composition essentially pins down the *location* where the transition occurs. Locations provide a natural way of identifying independence between transitions.

An example of this is shown in Figure 5.1 taken from [MN92], where the behaviour of the processes $a||b$ and $ab+ba$ are displayed as labelled transition systems. Notice that the transition system for $a||b$ has four transitions, but only two distinct labels on the transitions. This captures the fact that there are only two underlying (independent) events, one labelled a and the other labelled b. In contrast, the process $ab + ba$ has four distinct events.

It can be shown that the labels of the transition systems obtained by the above operational semantics follows the following fact.

Proposition 5.2.2 *For any transition $\hat{P} \xrightarrow[u]{\mu} \hat{P}'$, u is of the form*

1. *$s[P][P']$, where $s \in (\{0,1\} \cup \Lambda)^*$ and $\mu \in Act$.*
2. *$s[P][P']$, where $s \in (\{0,1\} \cup \Lambda)^*$ and $\mu = \tau$.*
3. *$s\langle s_0[P_0][P_0'], s_1[P_1][P_1']\rangle$, where $s, s_0, s_1 \in (\{0,1\} \cup \Lambda)^*$, and $\mu = \tau$.*

For convenience, the brackets around the process expressions in the labels are omitted so for example $s[P][P']$ is written sPP' instead.

Each distinct label in the transition system corresponds to an event, as follows.

Definition 5.2.3 (Events) *Define the set of events Ev as follows:*

$$Ev = \{(\mu, u) | \exists P, P' \in Proc. P \xrightarrow[u]{\mu} \}$$

For $e \in Ev$ we can identify $Loc(e) \subseteq \{0,1\}^*$, the location(s) where e occurs, as follows:

$$\forall e = (\mu, u).Loc(e) = \begin{cases} \{s \downarrow_{\{0,1\}}\} & \text{if } u = sPP' \\ \{ss_0 \downarrow_{\{0,1\}}, ss_1 \downarrow \{0,1\}\} & \text{if } u = s\langle s_0 P_0 P_0', s_1 P_1 P_1'\rangle \end{cases}$$

where for $s \in (\{0,1\} \cup \Lambda)^*, s \downarrow_{\{0,1\}}$ denotes the projection of s onto $\{0,1\}$. In other words, $s \downarrow_{\{0,1\}}$ is the subsequence of s obtained by erasing al elements not in $\{0,1\}$.

From the way the information about locations is introduced into the event labels, it is clear that the locations $Loc(e)$ of an event e is a string which identifies the nested component where e occurs. The independence relation is defined on locations and then lifted to events as follows:

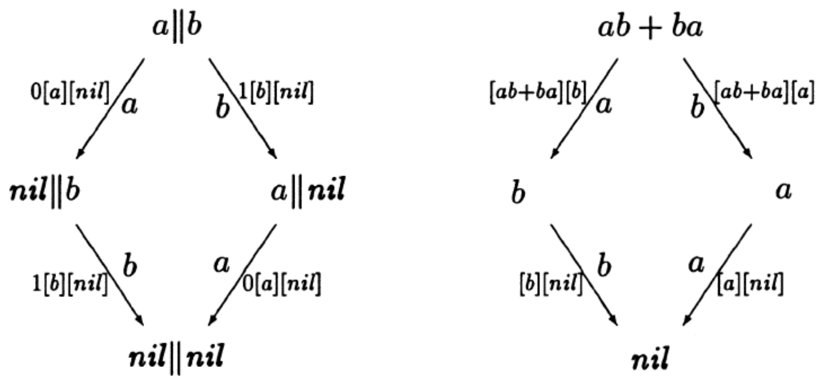

Figure 5.1: Concurrency versus non-deterministic interleaving

Definition 5.2.4 (Independence on events) *Define an* independence relation on locations $I_l \subseteq \{0,1\}^* \times \{0,1\}^*$ *as follows:*

$$\forall s, s' \in \{0,1\}^*.(s, s') \in I_l \textit{ iff } s \npreceq s' \wedge s' \npreceq s$$

where $\preceq$ *is the prefix relation on strings.*
We can extend this to a relation $\hat{I}_l \subseteq (\{0,1\} \cup \Lambda) * \times (\{0,1\} \cup \Lambda)^*$ *in the obvious way.*

$$\forall \hat{s}, \hat{s}' \in (\{0,1\} \cup \Lambda)^*.(\hat{s}, \hat{s}') \in \hat{I}_l \textit{iff } (\hat{s} \downarrow_{\{0,1\}}, \hat{s}' \downarrow_{\{0,1\}}) \in I_l$$

For convenience, we shall write both $\hat{I}_l$ *and* I_l *as* I_l.
Using I_l *we can define an* independence relation on events $I \subseteq Ev \times Ev$ *as follows:*

$$\forall e, e' \in Ev : (e, e') \in I \textit{ iff } \forall s \in Loc(e), \forall s' \in Loc(e') : (s, s') \in I_l$$

.

Having defined the sets of events Ev and the independence relation I on events, they define an asynchronous transition system corresponding to the operational behaviour of a process P. This is the transition system $TS_{CCS} = (S, i, E, I, Tran)$ where

- $S = \{[P] | P \in Proc\}$.
- $E = Ev$.
- $I \subseteq Ev \times Ev$ is given in Def 5.2.4.
- $Tran \subseteq S \times Ev \times S = \{([P], (\mu, u), [P'] =) | P \underset{u}{\xrightarrow{\mu}} P'\}$.

5.2.3 Denotational approach to giving event structure semantics to Pi-calculus

In the study of concurrent and distributed systems, the true concurrent semantics approach takes concurrency as a primitive concept rather than reducing it to non-deterministic interleaving. One of the by-products of this approach is that the casual links between process actions are more faithfully represented in true concurrent models.

Prime event structures [NPW81] are a casual model for concurrency which is particularly suited for the traditional process calculi such as CCS and CPS since they directly represent causality and concurrency simply as a partial order and an irreflexive binary relation. Winskel [Win82] proposed a compositional event structure semantics of CCS, that has been proved to be operationally adequate with respect to the standard labelled transition semantics.

The main issues when dealing with the full pi-calculus over CCS are name passing and the extrusion of bound names. These two ingredients are the source of the expressiveness of the calculus, but they are problematic in that they allow complex forms of casual dependencies.

Compared to a pure CCS, (either free of bound) name passing adds the ability to dynamically acquire new synchronization capabilities. For instance consider the pi-calculus process $P = n(z).(\overline{z} < a > |m(x))$, that reads from the channel n and uses the received name to output the name a in parallel with a read action on m. Hence a synchronization along the channel m is possible if a previous communication along the channel n substitutes the variable z exactly with the name m. Then in order to be compositional, the semantics of P must also account for "potential" synchronizations that might be activated by parallel compositions, like the one on channel m.

Casual dependencies in pi-calculus processes arise in two ways [BS98, DP99] by either nesting prefixes (called *subject* causality) or by using a name that has been bound by a previous action (called *object* causality). While subject causality is already present in CCS, object causality is distinctive to the pi-calculus. The interaction between the two forms of causal dependencies are quite complex.

This is especially understood in terms of the two ingredients of extrusion: name restriction and communication.

1. The restriction $(\nu n)P$ adds to the semantics of P a casual dependence between *every* action with subject n and *one of* the outputs with object n.

2. The communication of a restricted name adds *new* casual dependencies since both new extruders and new actions that need an extrusion may be generated by the variable substitution.

A casual semantics for pi-calculus should account for such a dynamic additional objective causality introduced by scope extrusion. In particular, the first item above hints at the fact that we have to deal with a form of disjunctive (objective) causality. Prime event structures are stable models that represent disjunctive causality by duplicating events and so that different copies causally depend on different (alternative) events. In this case it amounts to represent different copies of any action with a bound subject, each one causally depending on different (alternative) extrusions. However, the fact that the set of extruders dynamically changes, complicates the picture since new copies of any action with a bound subject should be dynamically spawned for each new extruder. In this way the technical details quickly becomes intractable.

The paper of Crafa, Varacca, and Yoshida [CVY12] follows a different approach, that leads to a nice technical development. The idea is to represent the disjunctive objective causality in a so-called inclusive way: in order to trace the causality introduced by scope extrusion it is sufficient to ensure that whenever an action with a bound subject is executed, at least one extrusion of that bound name must already executed, but it is not necessary to record which output was the real extruder. Clearly, such an inclusive-disjunctive causality is no longer representable with stable structures like prime event structures. However, it is shown that an operational adequate true concurrent semantics of the pi-calculus can be given by encoding a pi-process simply as a pair (E, X) where E is a prime event structure X is a set of (bound) names. Intuitively, the casual relation of E encodes the structural causality of a process. Instead, the set X affects the computation on E: where it is defined a notion of *permitted configurations*, ensuring that any computation that contains an action whose subject is a bound name in X, also contains a previous extrusion of that name. Hence a further benefit of this semantics is that it clearly accounts for both forms of causality: subjective causality is captured by the causal relation of event structures, while objective causality is implicitly captured by permitted configurations.

5.3 Preliminary results

Taking the work of Mukund and Nielsen [MN92], specially the way they add locations, we look into how this method would work for adding locations to the operational semantics of psi-calculi [BJPV11]. Preliminary results suggest that this can be done in an intuitive way, where we add the location labels to the tags of the transitions in psi-calculi in very much the same manner as Mukund and Nielsen did.

The operational semantics that we currently are working on and trying to prove is effectively giving the locations of processes is defined as bellow:

$$\frac{\Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd \underline{M}(\lambda\tilde{y})N.P \xrightarrow[{[\underline{M}(\lambda\tilde{y})N.P][P]}]{\underline{K}N[\tilde{Y}:=\tilde{L}]} P[\tilde{y} := \tilde{L}]} \text{ (IN)}$$

$$\frac{\Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd \overline{M}N.P \xrightarrow[{[\overline{K}N][P]}]{\overline{K}N} P} \text{ (OUT)}$$

$$\frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \textbf{case}\ \tilde{\varphi}\ :\ \tilde{P} \xrightarrow[{[\textbf{case}\ \tilde{\varphi}:\tilde{P}][P']}]{\alpha} P'} \text{ (CASE)}$$

$$\frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow[u]{\overline{M}(\nu\tilde{a})N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow[v]{\underline{K}N} Q' \qquad \Psi_Q \otimes \Psi_P \otimes \Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd P \,|\, Q \xrightarrow[\langle 0u, 1v\rangle]{\tau} (\nu\tilde{a})(P' \,|\, Q')} \text{ (LCOM)}$$

$$\frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow[u]{\overline{M}(\nu\tilde{a})N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow[v]{KN} Q' \qquad \Psi_Q \otimes \Psi_P \otimes \Psi \vdash M \dot{\leftrightarrow} K}{\Psi \rhd Q \,|\, P \xrightarrow[\langle 0v, 1u\rangle]{\tau} (\nu\tilde{a})(Q' \,|\, P')} \text{ (RCOM)}$$

$$\frac{\Psi \otimes \Psi_Q \rhd P \xrightarrow[u]{\alpha} P' \qquad bn(\alpha)\#Q}{\Psi \rhd P \,|\, Q \xrightarrow[0u]{\alpha} P' \,|\, Q} \text{ (LPAR)}$$

$$\frac{\Psi \otimes \Psi_Q \rhd P \xrightarrow[u]{\alpha} P' \qquad bn(\alpha)\#Q}{\Psi \rhd Q \,|\, P \xrightarrow[1u]{\alpha} Q \,|\, P'} \text{ (RPAR)}$$

$$\frac{\Psi \rhd P \xrightarrow[u]{\alpha} P' \qquad b\sharp\alpha, \Psi}{\Psi \rhd (\nu b)P \xrightarrow[(\nu b)u]{\alpha} (\nu b)P'} \text{ (SCOPE)}$$

$$\frac{\Psi \rhd P \xrightarrow[u]{\overline{M}(\nu\tilde{a})N} P' \qquad b\sharp\tilde{a}, \Psi, M \qquad b \in n(N)}{\Psi \rhd (\nu b)P \xrightarrow[u]{\overline{M}(\nu\tilde{a}\cup\{b\})N} P'} \text{ (OPEN)}$$

$$\frac{\Psi \rhd P|!P \xrightarrow[u]{\alpha} P'}{\Psi \rhd !P \xrightarrow[u]{\alpha} P'} \text{ (REP)}$$

The main two differences between what Mukund and Nielsen did is the way bounded names are marked. This we handle very similar though, and that we have replication rule instead of recursion. This later we leave the actual location discovery to the rule of parallel composition that it builds upon.

With these operation rules we can go from psi-calculi processes to asynchronous transition systems in very much the same way that [MN92] went from CCS to asynchronous transition systems. There are minor differences related to the fact that in psi-calculi we work only with replication, instead of general recursion as [MN92] considers, and also we do not have structural congruence. These changes do not affect much the proofs of similar results that Mukund and Nielsen had in [MN92]. In particular, we are able to prove the fact that the asynchronous transition systems that are obtained by the non-interleaving semantics of psi-calculus are exactly those that can be associated one-to-one with event structures, which are called *elementary* in [MN92], and correspond to those investigated in [WN95, Sec.10.2] rather differently in correlation with Petri nets.

We define events to be the same as in Def 5.2.3, and independence relation to be the same as Def 5.2.4. With these two definitions we can see that we have the same events and independence relation as in [MN92], and the same asynchronous transition system they obtain in that paper. Our preliminary investigations show that the logic aspect and binding of names affect very little the concurrency aspect. These parts of psi-calculi are not present in CCS.

From the logic aspect we can see that we do not maintain any logical information in the labels and thus events. And the logic only affects what transitions are produced. As concurrency in this transition system is based on when events are independent this logic will not affect this part. So the logic does not decide what is independent or not, just what events that exist in the first place.

We also look at extra dependencies that come from name binding as identified in [CVY12]. Preliminary investigations into this show that the operational semantics handles all of these problems for us, either through transitions being blocked due to dependencies or that the independence relation does not accept them as independent.

Further work, besides detailing the results above, involves looking closer at the relations between the denotational approach of [CVY12, WN95] and our non-interleaving operational approach. This means that we would extend the work on pi-calculus from [CVY12] to psi-calculus and then give an operational adequacy result.

Another line of further investigation is to look at known instances of psi-calculi and see how our results give non-interleaving semantics to such instances as pi-calculus, spi-calculus, and cc-pi.

Further interesting results could come from looking at how previous work from Chapters 3 and 4 on encoding event structures and DCR graphs [HM10a] in psi-calculi instances conforms with the non-interleaving semantics we give here. In particular, in Chapter 3 we gave a psi-calculus instance and encoded event structures as such psi-processes, making a tight correlation between the concurrency in event structures and the interleaving diamonds of the interleaving semantics of psi-calculus. But through our work here we would obtain an event structure (i.e., equivalent to the elementary ATS) as the semantic object for the same psi-process that encodes also an event structure. This rises the question of how are these related.

Chapter 6

Conclusions

We have encoded the true concurrency models of prime event structures and DCR-graphs into corresponding instances of psi-calculi. For this we have made use of the expressive logic that psi-calculus provides to capture the causality and conflict relations of the prime event structures, as well as the relations of DCR-graphs. The computation in the concurrency models corresponds to reduction steps in the psi-processes. The more expressive model of DCR-graphs required us to make use of the communication mechanism of psi-calculi, whereas for event structures this was not needed. The data terms we sent were tuples of terms, capturing markings of DCR-graphs with a generation number attached to them.

For the encodings we also investigated some results meant to provide more confidence in their correctness. In particular, for event structures we also looked at action refinement as well as gave the syntactic restrictions that capture the psi-processes that exactly correspond to event structures. Besides providing correlations between the computations in the respective models, we also investigated how true concurrency is correlated to the interleaving diamonds in the encodings we gave.

These results have been presented in Chapters 3 and 4, while Chapter 2 gave background information for these. In more concise presentations the same results have been published at international workshops [Nor13, NPH14a, NPH14b].

The purpose of our investigations was to see how well the expressiveness of psi-calculi can accommodate the expressiveness of true concurrency models. And with the above results we will say we have managed to a certain degree to show that we capture the expressiveness of the true concurrency models we have looked at in psi-calculi. This is only inhibited by the fact that psi-calculi currently do not have a built in way to model liveness properties, and these would have to be defined independently for each instantiation outside the standard psi-calculi definitions.

As future work in that regard is the possibility to look into adding responses to psi, as done in [CHPW12] for Transition Systems with Responses. This is interesting from the fact that we currently do not have a way to show the accepting runs from DCR-graphs in dcrPsi. This could be defined based on the assertions of dcrPsi, as we know these assertions are modelling the markings, and the accepting states of DCR-graphs are defined on its marking. We would prefer a way that would give psi-calculi its own accepting states on the meta-model level instead.

Nevertheless, a discrepancy remains between the interleaving semantics based on SOS rules of psi-calculi, and the true concurrency nature of the two models we considered. We have started to look into creating a non-interleaving semantic for psi-calculi in Chapter 5. The plan is to finish this work as a next step after this thesis. Following this we would be interested in checking if this non-interleaving semantic would give us the independencies for event structures with its own independence relation, as we get from our independence relation in eventPsi.

Bibliography

[Ace94] Luca Aceto. A static view of localities. *Formal Asp. Comput.*, 6(2):201–222, 1994.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 104–115. ACM, 2001.

[AFV01] Luca Aceto, Wan J. Fokkink, and Chris Verhoef. Structural Operational Semantics. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, chapter 3. Elsevier, 2001.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.

[BCHK93] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. Observing localities. *Theor. Comput. Sci.*, 114(1):31–61, 1993.

[BCHK94] Gérard Boudol, Ilaria Castellani, Matthew Hennessy, and Astrid Kiehn. A theory of processes with localities. *Formal Asp. Comput.*, 6(2):165–200, 1994.

[Bed88] Marek A. Bednarczyk. *Categories of asynchronous systems*. PhD thesis, Univ. Sussex, 1988.

[Ben12] Jesper Bengtson. Psi-calculi in isabelle. *Archive of Formal Proofs*, 2012, 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 *SEFM*, pages 74–89, 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 Rocco De Nicola, editor, *16th European Symposium on Programming Programming Languages and Systems (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2007.

[BS98] Michele Boreale and Davide Sangiorgi. A fully abstract semantics for causality in the π-calculus. *Acta Inf.*, 35(5):353–400, 1998.

[Che95] Allan Cheng. Petri nets, traces, and local model checking. In *AMAST*, volume 936 of *Lecture Notes in Computer Science*, pages 322–337. Springer, 1995.

[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.

[CVY07] Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. Compositional event structure semantics for the internal *pi* -calculus. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2007.

[CVY12] Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. Event structure semantics of parallel extrusion in the pi-calculus. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7213 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2012.

[dBdRR89] Jaco W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors. *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *Lecture Notes in Computer Science*. Springer, 1989.

[DFM$^+$05] Rocco De Nicola, Gian Luigi Ferrari, Ugo Montanari, Rosario Pugliese, and Emilio Tuosto. A process calculus for qos-aware applications. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *7th International Conference on Coordination Models and Languages*, volume 3454 of *LNCS*, pages 33–48. Springer, 2005.

[DP99] Pierpaolo Degano and Corrado Priami. Non-interleaving semantics for mobile processes. *Theor. Comput. Sci.*, 216(1-2):237–270, 1999.

[GM12] Eric Goubault and Samuel Mimram. Formal relationships between geometrical and classical models for concurrency. *Electronic Notes in Theoretical Computer Science*, 283:77 – 109, 2012.

[Gou93] Eric Goubault. Domains of Higher-Dimensional Automata. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 1993.

[Gup94] Vincent Gupta. *Chu Spaces: A Model of Concurrency*. PhD thesis, Stanford University, 1994.

[HM10a] Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In Kohei Honda and Alan Mycroft, editors, *3rd Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES)*, volume 69 of *EPTCS*, pages 59–73, 2010.

[HM10b] Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *PLACES*, volume 69, pages 59–73, 2010.

[HMS12] Thomas T. Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Nested dynamic condition response graphs. In Farhad Arbab and Marjan Sirjani, editors, *4th IPM International Conference on Fundamentals of Software Engineering (FSEN)*, volume 7141 of *LNCS*, pages 343–350. Springer, 2012.

[Maz88] Antoni W. Mazurkiewicz. Basic notions of trace theory. In de Bakker et al. [dBdRR89], pages 285–363.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge Univ. Press, 1999.

[MN92] Madhavan Mukund and Mogens Nielsen. Ccs, location and asynchronous transition systems. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*, volume 652 of *Lecture Notes in Computer Science*, pages 328–341. Springer, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i-ii. *Information and Computation*, 100(1):1–77, 1992.

[Nor13] Håkon Normann. Event Structures as Psi-calculi. In Tarmo Uustalu and Juri Vain, editors, *25th Nordic Workshop on Programming Theory (NWPT13)*, pages 53–56, 2013.

[NPH14a] Håkon Normann, Cristian Prisacariu, and Thomas Hildebrandt. Concurrency models with causality and events as psi-calculi. In Ivan Lanese and Ana Sokolova, editors, *7th Interaction and Concurrency Experience (ICE 2014)*, Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2014. (15 pages).

[NPH14b] Håkon Normann, Cristian Prisacariu, and Thomas Hildebrandt. True concurrency semantics for psi-calculi. In *1st International Workshop on Meta Models for Process Languages (MeMo)*, Berlin, Germany, June 2014. (6 pages).

[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.

[NPW81] Nielsen, Plotkin, and Winksel. Petri nets, event structures and domains, part I. *TCS: Theoretical Computer Science*, 13:85–108, 1981.

[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.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Univ. Aarhus, 1981.

[Pra91] Vaughan R. Pratt. Modeling concurrency with geometry. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA*, pages 311–322, January 1991.

[Pra95] Vaughan R. Pratt. Chu spaces and their interpretation as concurrent objects. In *Computer Science Today: Recent Trends and Developments*, 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.

[Pra02] Vaughan R. Pratt. Event-State Duality: The Enriched Case. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR'02*, volume 2421 of *LNCS*, pages 41–56. Springer, 2002.

[San96] Davide Sangiorgi. Locality and interleaving semantics in calculi for mobile processes. *Theor. Comput. Sci.*, 155(1):39–83, 1996.

[Shi85] M. W. Shields. Concurrent machines. *Computer Journal*, 28(5):449–465, 1985.

[SMHM13] Tijs Slaats, Raghava Rao Mukkamala, Thomas T. Hildebrandt, and Morten Marquard. Exformatics declarative case management workflows as dcr graphs. In Florian Daniel, Jianmin Wang, and Barbara Weber, editors, *11th International Conference on Business Process Management (BPM)*, volume 8094 of *LNCS*, pages 339–354. Springer, 2013.

[vG06] Rob van Glabbeek. On the Expressiveness of Higher Dimensional Automata. *Theor. Comput. Sci.*, 356(3):265–290, 2006.

[vGG01] Rob J. van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001.

[vGP09] Rob J. van Glabbeek and Gordon Plotkin. Configuration structures, event structures and Petri nets. *Theor. Comput. Sci.*, 410(41):4111–4159, 2009.

[Win82] Glynn Winskel. Event structure semantics for ccs and related languages. In Mogens Nielsen and Erik Meineche Schmidt, editors, *9th Colloquium on Automata, Languages and Programming (ICALP)*, volume 140 of *LNCS*, pages 561–576. Springer, 1982.

[Win86] Glynn Winskel. Event structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *LNCS*, pages 325–392. Springer, 1986.

[Win88] Glynn Winskel. An introduction to event structures. In de Bakker et al. [dBdRR89], pages 364–397.

[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 – vol 4 – Semantic Modelling*, pages 1–148. Oxford Univ. Press, 1995.