Journal of Logical and Algebraic Methods in Programming 85 (2016) 368–398

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Declarative event based models of concurrency and refinement in psi-calculi

Håkon Normann [a,*,1], Christian Johansen [b,2], Thomas Hildebrandt [a,1]

[a] IT University of Copenhagen, Rued Langgaardsvej 7, 2300 Copenhagen, Denmark
[b] Dept. of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway

ARTICLE INFO

Article history:
Received 31 October 2014
Received in revised form 27 November 2015
Accepted 23 December 2015
Available online 6 January 2016

Keywords:
Psi-calculi
Prime event structures
DCR graphs
Action refinement
Declarative models
Concurrency

ABSTRACT

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

1. Introduction

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

The present work is part of a more ambitious research goal pursued in the CompArt project.[3] The final aim is to provide a foundation for the description and formal reasoning of adaptable, distributed and mobile computational artefacts under regulative control.

* Corresponding author. Tel.: +45 7218 5000.
E-mail addresses: hnor@itu.dk (H. Normann), cristi@angeloti.info (C. Johansen), hilde@itu.dk (T. Hildebrandt).

[1] The first and last authors are supported by the Velux Foundation through the COMPART project (grant 33295).
[2] The second author (née Cristian Prisacariu) was partially supported by the project OffPAD (https://www.offpad.org) with number 8324 part of the Eurostars program (https://www.eurostars-eureka.eu/project/id/8324) funded by the EUREKA and Norwegian Research Council (http://www.forskningsradet.no/prognett-eureka/Forside/1224698203622).
[3] Computational Artefacts (CompArt), http://www.compart.ku.dk.

http://dx.doi.org/10.1016/j.jlamp.2015.12.007

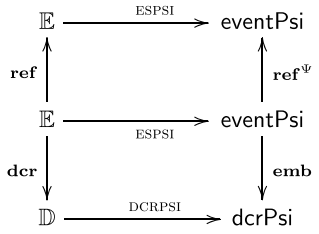

Fig. 1. Summary of the results in this paper, where: $\mathbb{E}$ is the class of finite prime event structures, $\mathbb{D}$ is the class of DCR graphs, eventPsi and dcrPsi are the two psi-instances we define, **ref** is the refinement operation of [17], $\mathbf{ref}^{\Psi}$ is the corresponding refinement operation we define for eventPsi processes, and **dcr** and **emb** are embeddings of event structures and eventPsi into DCR graphs respectively dcrPsi.

A number of proposals have been made for concrete formal models and reasoning techniques for distributed and mobile systems supporting different kinds of adaptability (e.g. [1–5]). The plethora of models indicates that our understanding of the needs for such agile and adaptable computational artefacts is still developing. For this reason, we aim for a foundation that facilitates experimentation with different formal process calculi. Our focus on mobility, shared resources, regulative control and adaptability leads us to consider the formal meta process calculus of psi-calculi [6]. Psi-calculi provides a general setting for the definition of process calculi for distributed and mobile processes, that generalises the seminal pi-calculus [7] in two dimensions: *(i)* generalisation of channel names to nominal data structures [8], i.e. general terms with a notion of local names; and *(ii)* a general logic for expressing constraints guarding actions. Examples of calculi encoded as psi-calculi include the spi- and applied-pi calculi [9,10] and the CC-pi [11,12].

While the general nominal data structures and channel communication provide a foundation for expression of shared mobile resources, the constraint logic provides a foundation for declarative expression of control and regulations. Concretely, we show how two declarative, event-based process models for concurrency, i.e., the seminal prime event structures [13,14] and the Dynamic Condition Response (DCR) Graphs [15], can be represented as psi-calculi. We consider declarative event-based concurrent models for two reasons: Firstly, declarative models more naturally describe regulations, that is, rules governing processes. Secondly, they are well-behaved with respect to action refinement [16,17], which we see as a fundamental step towards supporting agile development and adaptation. Simply put, action refinement is the method of developing a system by starting with an abstract specification, and gradually refining its components (or actions) by providing more details. Thus an action can be changed from being instantaneous to having structure, or duration. This should not be confused with the notion of refinement often found in process algebras where an implementation refines a specification by reducing the set of execution traces.

Fig. 1 gives an overview of the main results from this paper. After providing the necessary background on psi-calculi in Section 2, we give a representation of model of finite prime event structures [13,14] as an instance of psi-calculi in Section 3. The encoding function ESPSI, illustrated by the middle horizontal arrow in Fig. 1 exploits the logic of psi-calculi to represent the causality, independence and conflict relation of event structures. This allows us to prove that the representation respects the semantics up to concurrency diamonds and action refinement [17]. Concretely, we define action refinement $\mathbf{ref}^{\Psi}$ on the eventPsi-processes, and prove in Sec. 3.3 that also action refinement is preserved by our translation, making the upper square diagram of Fig. 1 commute.

We also identify the syntactic shape of psi-processes that correspond to finite prime event structures. This last result can be seen as a characterisation of the psi-processes for which the middle arrow in Fig. 1 is one part of an equivalence.

Event structures are a denotational model and cannot provide finite representations of infinite behaviours. To accommodate finite representations of infinite behaviours, we proceed in Section 4 to consider Dynamic Condition Response graphs (abbreviated DCR graphs or just DCRs) [15]. DCR graphs are an event-based model of concurrency strictly generalising event structures by permitting the events to happen more than once (as opposed to in event structures, where events happen at most once) and by refining the relations of dependency and conflict between events. This generalises event structures in two ways: Firstly, DCR graphs are a so-called system model allowing finite representations of infinite behaviours. Secondly, DCR graphs allow representation of acceptance criteria for computations, making it possible to express both safety and liveness properties. We will not consider acceptance criteria in the present paper. DCRs have been shown to support run-time adaptation [5] and have been successfully used in industry to model and support flexible and adaptable business processes for knowledge workers [18,19].

As for event structures, we provide an encoding DCRPSI of DCR graphs into the psi-instance dcrPsi, shown in Fig. 1 as the lower arrow. Again we identify the syntactic shape of those dcrPsi-processes which corresponds exactly to DCR graphs, i.e. characterises the dcrPsi-processes for which the mapping is part of an equivalence, and prove a bisimulation relation between the DCR graphs semantics and their encoding for a naturally defined event-labelled transition system semantics of the encoding.

We end the section by showing that the encoding of DCR graphs in dcrPsi is a conservative generalisation of the encoding of finite event structures by showing that the lower square diagram in Fig. 1 commutes, for the standard embedding **dcr** of event structures into DCR graphs and a suitably defined, semantics preserving embedding **emb** of eventPsi-processes (corresponding to event structures) into dcrPsi-processes (corresponding to DCR graphs).

We end in Section 5 by concluding and outlining the path towards the final aim of this research, which is to explore nominal calculi for declarative, run-time adaptable mobile processes with shared resources, subject to both safety and liveness regulations.

This paper extends [20] by including new results, more examples, motivation and background. In particular, the background section on psi-calculi has been considerably enlarged, including more definitions, examples and intuitions. The same was done for the backgrounds on DCR graphs and event structures and refinement. The results have more detailed proofs and include new results that complete the diagram of Fig. 1 with the embedding of eventPsi into dcrPsi, as well as the syntactic restrictions for dcrPsi-processes that make the lower arrow part of an equivalence.

2. Background on psi-calculi

Psi-calculi [6] have been developed as a framework for defining nominal process calculi, like the many variants of the pi-calculus [7]. Instances of psi-calculi have been made for applied-pi calculus [10] and for CC-pi [11,12] which can in turn capture probabilistic models. Typed psi-calculi exist [21] as well as the related instance [22] for distributed pi-calculus [23,24].

The psi-calculi framework is based on nominal datatypes. We assume an infinite set of atomic *names* $\mathcal{N}$ ranged over by $a, b, \ldots$. Intuitively, names are symbols that can be statically scoped, as well as be subjected to substitution (which we define later). A nominal datatype is then constructed from a nominal set [8], which is a set equipped with *name swapping* functions, written (ab), satisfying certain natural axioms such as $(ab)((ab)t) = t$. Intuitively, applying the name swapping (ab) changes a term t by replacing a with b and b with a. One main point is that even without having any particular syntax for constructing t we can define what it means for a name to "occur" in a term, i.e., when the term can be affected by a swapping that involves that name. The names occurring in this way in a term t constitute the *support* of t, written $n(t)$. In usual datatypes, without binders, we will have $a \notin n(t)$ if a does not occur syntactically in t. Whereas in the lambda calculus the support corresponds to the free names, since terms are identified up to alpha-equivalence. A function f is *equivariant* if $(ab)\,f(t) = f((ab)t)$ holds for all t. We can then define a nominal datatype formally as follows.

Definition 2.1 *(Nominal datatypes and substitutions).* A *nominal datatype* is a nominal set together with a set of equivariant functions on it. Psi-calculi consider *substitution functions* that substitute terms for names. If t is a term of a datatype, $\tilde{a}$ is a sequence of names without duplicates, and $\tilde{T}$ is an equally long sequence of terms of possibly different datatypes, the *substitution* $t[\tilde{a} := \tilde{T}]$ is a term of the same datatype as t. The only formal requirements for substitutions are that a substitution is an equivariant function that satisfies two substitution laws:

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

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

Definition 2.2 *(Parameters).* The psi-calculi framework is *parametric*; instantiating the parameters accordingly, one obtains an *instance of psi-calculi*, like the pi-calculus, or the cryptographic spi-calculus. These parameters are:

T	terms (data/channels)
C	conditions
A	assertions

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

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

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

Intuitively, terms can be seen as generated from a signature, as in term algebras [25]. We can think of the conditions and assertions like in first-order logic: the minimal assertion being top/true, entailment the one from first-order logic, and composition taken as conjunction. It is helpful to think of assertions and conditions as logical formulas, and the entailment

relation as an entailment in logic; but allow the intuition to think of logics abstractly, not just propositional logic, so that assertions and conditions are used to express any logical statements, where the entailment defines when assertions entail conditions (do not restrict to only thinking of truth tables; e.g., in our encodings we will use an extended logic for sets, with membership, pairs, etc.). The intuition of entailment is that $\Psi \vdash \varphi$ means that given the information in Ψ, it is possible to infer φ. Two assertions are equivalent if they entail the same conditions.

Definition 2.3 *(Assertion equivalence).* Two assertions are *equivalent*, written $\Psi \simeq \Psi'$, iff for all φ we have that $\Psi \vdash \varphi \iff \Psi' \vdash \varphi$.

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

Definition 2.4 *(Requisites on valid psi-calculus parameters).* The following properties must be satisfied by any psi-instance.

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

Channel equality is a *partial equivalence* which means that there can be terms that are not equivalent with anything (not even themselves). This does not allow them to be used as channels (but only as data). The composition of assertions (wrt. assertion equivalence) must be associative, commutative, and have **1** as unit; moreover, composition must preserve equivalence of assertions.[4,5]

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

Definition 2.5 *(Syntax).* The syntax for building psi-processes is the following (psi-processes are denoted by $P, Q, \ldots$; terms from **T** by $M, N, \ldots$):

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

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

The *input* and *output* processes are as in pi-calculus except that the channel objects M can be arbitrary terms. In the input process the object $(\lambda\tilde{x})N$ is a pattern with the variables $\tilde{x}$ bound in N as well as in the continuation process P.[6]

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

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

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

Intuitively, any term message received on M must match the pattern N for some substitution of the variables $\tilde{x}$. The same substitution is used to substitute these variables in P after a successful match. The traditional pi-calculus input $a(x).P$ would be modelled in psi-calculi as $\underline{a}\langle(\lambda x)x\rangle.P$, where the names are the only terms allowed. Restriction, parallel composition, and replication are the standard constructs of pi-calculus.

The **case** process behaves like one of the P_i for which the condition φ_i is entailed by the current environment assumption, as defined by the notion of *frame* which we present later. Frames are familiar from the applied pi-calculus [10], where were introduced with the purpose of capturing static information about the environment (or seen in reverse, the frame is the static information that the current process exposes to the environment). Particular examples of using the case construct are:

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

Remark 2.6. Psi-calculi work with finite terms and processes. Therefore, we restrict our further investigations to finite event structures and DCRs. To handle event structure over an infinite set of events one needs to investigate extensions of psi-calculi with three forms of infinity:

1. Infinite summation, which is sometimes found in process algebras, e.g., in Milner's SCCS [26]. In the case of psi-calculi an infinite case construct can be written as **case** $\tilde{\varphi}_i : \tilde{P}_i$ where infinite lists are used to represent the respective condition/process pairs. No significant changes to the semantics would be needed.
2. Infinite parallel composition could use the same semantic rule as for the finite case, but care needs to be taken with the required notions of frame and entailment. Often the replication is the preferred way to obtain infinite parallel components.
3. Infinite nominal data structures, where works into infinite terms would be a starting point.

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

Example 2.7 *(Pi-calculus as an instance).* To obtain pi-calculus [7] as an instance of psi-calculi use the following, built over a single set of names $\mathcal{N}$:

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

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 2.8. From the instance created in Example 2.7 one can obtain the polyadic pi-calculus [27] by adding tupling symbols t_n for tuples of arity n to $\mathbf{T}$, i.e.

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

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

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

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

Psi-calculi are given an operational semantics in [6] using labelled transition systems, where the states are the process terms and the transitions represent one reduction step, labelled with the action that the process executes. The actions, generally denoted by α, β, represent respectively the input and output constructions, as well as τ the internal synchronisation/communication action:

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

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

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

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

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

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

Definition 2.9 *(Frame).* A *frame* is of the form $(\nu\tilde{b})\Psi$ where $\tilde{b}$ is a sequence of names that bind into the assertion Ψ. We write just Ψ for $(\nu\epsilon)\Psi$ when there is no risk of confusing a frame with an assertion. We identify alpha variants of frames. In consequence, composition of frames is defined by $(\nu\tilde{b_1})\Psi_1 \otimes (\nu\tilde{b_2})\Psi_2 = (\nu\tilde{b_1}\tilde{b_2})\Psi_1 \otimes \Psi_2$ where $\tilde{b_1} \notin n(\tilde{b_2}, \Psi_2)$ and vice versa. The frame of a process $\mathcal{F}(P)$ is defined inductively on the structure of the process as:

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

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

Example 2.10. Calculating the frame of the following process, when $a \notin n(\Psi_1)$, is:

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

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

Definition 2.11 *(Semantics).* The transition rules for psi-calculi are the following, where the symmetric rules for (PAR) and (COM) are elided.

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

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

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

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

$$\frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \varphi_1 : P_1, \ldots, \varphi_n : P_n \xrightarrow{\alpha} P'}\ (\text{CASE}) \qquad \frac{\Psi \otimes \Psi_Q \rhd P \xrightarrow{\alpha} P' \qquad bn(\alpha) \notin n(Q)}{\Psi \rhd P \parallel Q \xrightarrow{\alpha} P' \parallel Q}\ (\text{PAR})$$

$$\frac{\Psi \rhd P \xrightarrow{\alpha} P' \qquad b \notin n(\alpha), b \notin n(\Psi)}{\Psi \rhd (\nu b)P \xrightarrow{\alpha} (\nu b)P'}\ (\text{SCOPE}) \qquad \frac{\Psi \rhd P \parallel !P \xrightarrow{\alpha} P'}{\Psi \rhd !P \xrightarrow{\alpha} P'}\ (\text{REP})$$

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

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

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

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

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

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

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

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

Example 2.12. For a simple example of a transition, suppose for an assertion Ψ and a condition φ that $\Psi \vdash \varphi$. Also assume that

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

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

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

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

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

2.1. Terms

A more detailed introduction to nominal sets used in psi-calculi can be found in [6, Sec. 2.1] and the recent book [8] contains a thorough treatment of both the theory behind nominal sets as well as various applications, e.g., see [8, Ch. 8] for nominal algebraic datatypes. For our presentation here we expect only some familiarity with notions of algebraic datatypes and term algebras. In the following we briefly present the notion of terms that we will be using in our encodings in the rest of the paper.

Definition 2.13 *(Terms, cf. [25, Chap. 3.1]).* In universal algebra, *terms* are constructed from a signature $\mathcal{F}$ of function names of some arity, and a set of variables $\mathcal{X}$, and are denoted $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Function symbols of arity 0 are called constants. Function symbols are sometimes denoted as $f(\_)$ or $f(\_,\_)$ to emphasise their arity (i.e., number of arguments, respectively one and two in this example). Terms without variables are called *ground*, their set being denoted by $\mathcal{T}(\mathcal{F})$, whereas terms containing variables are sometimes called *open*, to emphasise this aspect (in consequence, some works call ground terms *closed*). Every variable is also a term, i.e., an open term.

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

Example 2.14 *(Natural numbers as ground terms).* The set of natural numbers can be seen as terms denoted by $\mathbb{N}$ and defined as $\mathcal{T}(\{s(\_), 0\})$, i.e., only ground terms, built from the single constant term 0 using the unary successor function. An example of the term representing number 3 is $s(s(s(0)))$.

Definition 2.15 *(Multi-sorted terms, cf. [28]).* A multi-sorted algebraic structure is obtained if we add a notion of *sorts* to the Definition 2.13. Consider a set of sorts $s \in \mathcal{S}$ with a partial order on them, e.g., $s_1 < s_2$ means that any term of sort s_1 is also of sort s_2. Assign to each function symbol $f \in \mathcal{F}$ a sort for each parameter and a sort for output; e.g., $f : s_1, \ldots, s_n \to s$ takes n arguments of the respective sorts. In particular, each constant symbol is of some sort. One function can now be applied

only to terms of the appropriate sort, to produce a term of the respective result sort. The set of variables is now partitioned into sorted variables, i.e., each variable is of a certain sort. We sometimes mention the set of sorts as $\mathcal{T}(\mathcal{F}, \mathcal{X}, \mathcal{S})$ when we want to be specific; but most of the time we rely on the context for disambiguation.

Definition 2.16 *(Equality of terms). Equations* can be defined between terms, $t_l = t_r$, to express which terms should be viewed as equal. Equations are usually defined between open terms and are closed under substitutions, meaning that any two terms obtained by applying the same substitution to both t and t' would also be considered equal. For some set of equations, we infer the equality of two terms by applying the inference rules of equational logic, i.e., identity, symmetry, transitivity, closure under function application and under substitutions.

Example 2.17 *(Multi-sets as ground multi-sorted terms).* One representation of sets can be given as terms built over two sorts $\mathcal{S} = \{el, set\}$ using constant symbols e_i of sort *el* and constant symbol $\emptyset$ of sort *set*, and using a multi-sorted concatenation operation $\_ _{el} : \_ _{set}$ which takes the first argument of sort *el* and the second argument of sort *set*. An example is $e_1 : (e_2 : (e_3 : \emptyset))$. Because elements can appear several times in a concatenation, we have just modelled *multi-sets*. All multi-sets over some $e_i \in E$ are denoted by $\mathbb{N}^E$. With standard equations one could treat the multi-set terms with duplicate elements as equal to terms with a single copy of the element so that, e.g., $e_1 : (e_1 : \emptyset)$ would be equal to $e_1 : \emptyset$. Examples of equations of relevance to this case are: commutativity in multi-sets could be $e_1 : (e_2 : x) = e_2 : (e_1 : x)$, whereas an annihilation equation would be $e_1 : (e_1 : x) = e_1 : x$. We denote all the terms capturing sets over some E of constant symbols by 2^E.[7]

Example 2.18 *(Natural numbers with operations).* When interested in operations on natural numbers we could build a term algebra $\mathbb{N}_{op}$ from $\mathbb{N}$ of Example 2.14 by adding operators and equations related to the operators, as well as variables. Consider two sorts $\mathcal{S} = \{g, op\}$ and have all ground terms from Example 2.14 to be of sort g, i.e., put the constant 0 and the successor function to be of sort g. Take the order $g < op$, saying that any ground number g is also a number term with operations. Now define any operations, like $\_ + \_$ or $\_ - \_$, to be of sort *op*, i.e., taking as input *op* terms and returning *op* terms. Put the standard definition of such an operator into equations, e.g.: $s(0) + x = s(x)$. One can also put as equations the standard properties of such operators, like commutativity. So in our example $\mathbb{N}_{op} = (\{s, 0, +, -\}, \mathcal{X}, \mathcal{S})$. The sorts have been used just to make the representation nicer, i.e., having the natural numbers as building blocks on which the operations work. But in the presence of equations we can safely do without sorts; we will just have the successor function possibly applied to an operation like $s(s(0) + s(s(0)))$.

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

Example 2.19 *(Multi-sets with operations).* Take the sets and multi-sets of Example 2.17 and add functional symbols for standard operations like: $\_ \cup \_$, $\_ \setminus \_$, $\_ + \_$ (the last one standing for summation of multi-sets). Consider the definitions of these operations in the equations; for example $e_1 : e_2 : \emptyset \cup e_3 : \emptyset = e_1 : e_2 : e_3 : \emptyset$. Variables $\mathcal{X}$ are included as well. We could also add sorts for ground sets and sets with operators, as we did in the previous example. We denote such sets and multi-sets with operators and variables over some E by $2^E_{op} = (\{:, \emptyset, \cup, \setminus\}, \mathcal{X})$ and $\mathbb{N}^E_{op} = (\{:, \emptyset, +, -\}, \mathcal{X})$.

Many times data structures used in computer science are multi-sorted, and thinking in terms of sorts makes our results easier to follow. Therefore, we give a few definitions for sorts in the case when names are present. Complete treatment can be found in references like [8,29–31].

Definition 2.20 *(Multi-sorted nominal datatypes, cf. [8, ch. 8] or [29]).* Consider a set of *name sorts* $\mathcal{S}^{\mathcal{N}}$ disjoint from the set of sorts used for the datastructures, which we will call *data sorts* and denote $\mathcal{S}^{\mathcal{D}}$. Each name is assigned a name sort, the same as we were doing for variables. Name swapping is now *sort-respecting* in the sense that the two names being swapped must have the same name sort. In consequence, *freshness* and *name abstraction* are also sort-respecting (see [8, ch. 4.7]). Nominal datastructures are built over sorts described using the following grammar:

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

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

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

Our use of sorts in psi-calculi is rather simplistic, mainly to make sure that the right kind of terms are being used in the right place; e.g., when receiving data on a channel. We prefer to minimise mentioning sorting aspects, as for our results these details would mean too much cluttering without gained insights or correctness concerns. Nevertheless, when strictness is necessary, like when working with a proof assistant, then all details of the sorting should be in place, and the methods described in [30,31] should be followed. We give here a brief definition of some main aspects of sorted psi-calculi.

Definition 2.21 *(Sorted psi-calculi, [30, sec. 2.4]).* Multi-sorted psi-calculi use two main notions on top of multi-sorted nominal datatypes:

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

multi-sorted substitutions need to know which sorts (of terms) can substitute which sorts of names; i.e., a relation $<_{sub} \subseteq \mathcal{S}_{\mathcal{N}} \times \mathcal{S}$. The relation on sorts from Definition 2.15 is respected in the sense that if $a <_{sub} s_2$ and $s_1 < s_2$ then $a <_{sub} s_1$.

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

Example 2.22 *(Simple use of sorts in psi-instances).* Often a function like pairing is multi-sorted, $(\_,\_) : s_1 \times s_2 \to s_3$. The left parameter may be of sort natural numbers whereas the right parameter may be a multiset; everything could be considered of sort *pairs*. We would like to allow names to be used as parameters, and these names could be replaced upon a psi-communication. In this case we just say that the names must be of sort natural number and multiset. In consequence, the substitutions to respect the sorts should replace the name on the left only with natural number terms and the name on the right only with multisets.

3. Representing event structures in psi-calculi

In this section we provide a psi-calculus representation of finite *prime* event structures (recalled in Definition 3.1).

It is fairly easy to represent the interleaving, transition semantics for a finite event structure as a psi-calculus term. However, in contrast to most process calculi, event structures and more expressive event-based models of concurrency [32–39] come with a non-interleaving semantics. A non-interleaving semantics makes it possible to distinguish between interleaving and independence (sometimes called "true" concurrency) and are well behaved wrt. action refinement [17]. A simple example is given by two concurrent processes executing each a different instance of the same action a. An interleaving transition system based model would represent such a process by an "interleaving" diamond with all four sides labelled by the same action, which semantically typically would be equal to the sequential composition $a.a$ of the two actions. Refining the action a into $a1.a2$ in the semantical model, would thus result in the single sequence $a1.a2.a1.a2$ as the possible behaviour. However, when refining the parallel composition of two concurrent processes that both executes a, one would expect all possible interleavings, that is, the two different behaviours $\{a1.a2.a1.a2, a1.a1.a2.a2\}$.

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

We show in the subsequent section that this idea can be generalised to DCR graphs, and we believe that also other generalised versions of event structures [14,36] can be represented as psi-calculi following a similar approach as presented here.

3.1. Background on event structures

We follow the standard notation and terminology from [40, sec. 8].

Definition 3.1 *(Prime event structures). A labelled prime event structure* over alphabet Act is a tuple $\mathcal{E} = (E, \leq, \sharp, l)$ where E is a possibly infinite set of events, $\leq \subseteq E \times E$ is a partial order (the *causality* relation) satisfying

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

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

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

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

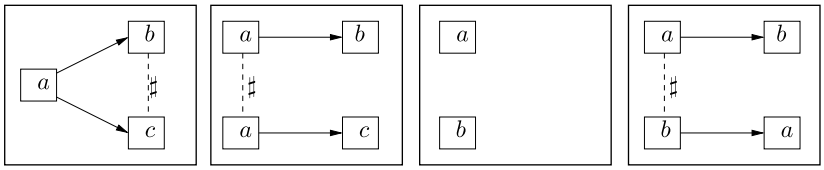

Fig. 2. Classic examples of event structures. Different boxes represent different events, possibly labelled the same. Arrows represent causality (not displaying those coming from the transitive closure); and dashed lines represent the symmetric conflicts (not displaying those coming from heredity).

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

Labels can be understood as actions, with a wide and general meaning. Events are instances of actions, and an action can happen several times, thus as different events. The same action can also happen in different components running in parallel, giving rise to *autoconcurrency*, as exemplified in the beginning of this section. Actions are important for observational equivalence, but not only them (see Example 3.5).

Example 3.2. In Fig. 2 we pictured four simple examples of finite event structures (taken from [41, Fig. 4]). We illustrate events as boxes containing their labels, the dependency relation by arrows, and the conflict relation by dashed lines with a $\sharp$ sign. In the left-most event structure we have three events, where the events labelled b and c depend on the event labelled a and are in conflict with each other. This is a standard branching point which could be specified in a simple CCS notation as $a;(b+c)$. In the second event structure we have two (conflicting) events, both labelled with a, and two events labelled b and c which depend on the first and the second a-labelled event respectively. Because of the principle of conflict hereditary, b is in conflict with the lower a-labelled event and similarly, c is in conflict with the upper a-labelled event (but the conflict relations are in this case usually not explicitly illustrated). In CCS notation this could be $a;b+a;c$. In the third event structure we have just two events without any explicit or inherited relation, which means that they are *concurrent* (e.g., $a \| b$), as made precise below. The last event structure is similar to the second, except that it offers two conflicting paths with the label a followed by b or b followed by a respectively.

Definition 3.3 *(Concurrency). Causal independence (concurrency)* between events is defined in terms of the above two relations as

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

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

From the definition it follows that only the two events in the third event structure in Fig. 2 are concurrent.

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

Definition 3.4 *(Configurations).* Define a *configuration* of an event structure $\mathcal{E} = (E, \leq, \sharp)$ to be a finite subset of events $C \subseteq E$ that respects:

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

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

Conflict-freeness is saying that no two conflicting events can happen in one run. This also says that once an event is discarded it can never be executed on the current run. This is similar to how the semantics of the choice operator in process algebras is defined (see rule (CASE) in Section 2) where all other branches of the choice are discarded once a step is taken. The downwards-closure says that all the dependencies of an executed event (i.e., which is part of a configuration) must have been executed also (on this same run).

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

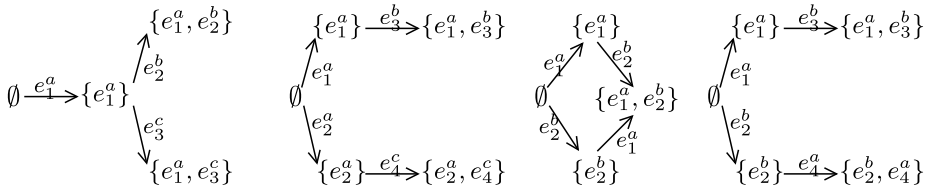

Fig. 3. Configurations and steps for event structures in Fig. 2. The labels of the events are shown as superscripts.

Example 3.5. For the examples from Fig. 2 we get the configurations and steps depicted in Fig. 3.

One may note that if only the paths of labels are observed, the two first event structures are indistinguishable, but if the branching structure is observed, i.e. by a bisimulation equivalence, they are distinguishable. One may also note that if the paths of labels are observed and even if branching time is observed, the two latter event structures are indistinguishable, but if concurrency is observed, i.e. by a history-preserving bisimulation equivalence [42,17], they are distinguishable.

Remark 3.6. It is known (see e.g., [40, Prop. 18]) that prime event structures are fully determined by their sets of configurations, i.e., the relations of causality, conflict, and concurrency can be recovered only from the set of configurations $\mathbb{C}_{\mathcal{E}}$ as follows:

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

It is also known (see e.g., [40, Sec. 8] for prime event structures or [17, Sec. 4] for the more general event structures of [14]) that there is no loss of expressiveness when working with finite, instead of infinite configurations. An infinite configuration can be obtained from infinite union of finite configurations coming from an infinite run.

For some event e we denote by $\leq e = \{e' \in E \mid e' \leq e\}$ the set of all events which are conditions of e (which is the same as the notation $\lceil e \rceil$ from [40], but we prefer to use the above so to be more consistent with similar notations we use in the rest of this paper for similar sets defined for DCRs too), and $\sharp e = \{e' \in E \mid e' \sharp e\}$ those events in conflict with e. We denote by $< e = \leq e \setminus \{e\}$ the non-trivial conditions of e, i.e., excluding itself.

3.2. The encoding of prime event structures

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

The intuition of the encoding is to represent event structure configurations as assertions and the causality and conflict relations as conditions. We do this by taking assertions $\mathbf{A}$ to be sets of events representing the history of executed events, i.e. a configuration, and composition just set union. Conditions $\mathbf{C}$ are taken to be pairs (p, c) of sets of events and entailment relation is then defined such that p represents the preconditions of an event, i.e. $\leq e$, and c represents the conflict set, i.e. $\sharp e$. That is, an assertion (history of executed events) Ψ entails a condition (p, c) if and only if $p \subseteq \Psi$ (the preconditions have been executed) and $c \cap \Psi = \emptyset$ (none of the conflicting events have been executed). Finally, we take the set of terms $\mathbf{T}$ to be simply a set of constants representing the events. To keep the exposition simple, we will ignore event labels until Section 3.3, where we treat action refinement. But labels could easily be added by redefining the terms to be pairs of an event and its label, for some given labelling function; e.g., $(e, l(e))$.

Definition 3.7 *(Event psi-calculus over E).* We define a psi-calculus instance, called eventPsi, parametrised by a set E of constant symbols, to be understood as *events*, by providing the following definitions of the key elements of a psi-calculus instance:

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

$$\mathbf{1} \stackrel{def}{=} \emptyset$$
$$\vdash \stackrel{def}{=} \begin{cases} \Psi \vdash (D, C) & \text{iff } (D \subseteq \Psi) \wedge (C \cap \Psi = \emptyset) \\ \Psi \vdash e \leftrightarrow f & \text{iff } e = f \end{cases}$$

where **T**, **C**, and **A** are algebraic data types built over the constants in E.

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

Remark 3.8. We are not using the nominal aspects of psi-calculi. Throughout the rest of this section we do not work with names, and therefore the support of all terms will be empty. Names will make their appearance in the encoding of DCRs in Section 4.

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

Definition 3.9 *(Event structures to* eventPsi*).* For $\mathcal{E} = (E, \leq, \sharp)$ an event structure and a configuration C of $\mathcal{E}$, define $\text{ESPSI}(\mathcal{E}, C)$ as

$$\text{ESPSI}(\mathcal{E}, C) = \|_{e \in E}\ P_e$$

with

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

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

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

We have seen that the eventPsi-processes that we obtain from event structures in Definition 3.9 have a specific syntactic form. But the eventPsi instance allows any process term to be constructed over the three nominal data-types that we gave in Definition 3.7. Below we give the syntactic restrictions on eventPsi-process terms corresponding to event structures, via the mapping defined by Theorem 3.19.

Definition 3.10 *(Syntactic restrictions for* eventPsi*).* We define an eventPsi-process to be *syntactically correct* if it is constructed using the grammar:

$$P_{ES} \quad := \quad (\!|\{e\}|\!) \quad | \quad \textbf{case}\ \varphi : \bar{e}\langle e \rangle.(\!|\{e\}|\!) \quad | \quad P_{ES} \parallel P_{ES}$$

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

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

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

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

Definition 3.11 *(Event transitions).* We define transitions between syntactically correct eventPsi processes P and P' to be $P \overset{e}{\rightsquigarrow} P'$ iff $\mathbf{1} \rhd P \xrightarrow{\bar{e}e} P'$.

Remark 3.12. Arbitrary eventPsi-processes can have different kinds of labelled transitions, but for syntactically correct processes the restrictions guarantee that only event transitions exist.

Lemma 3.13. *For a syntactically correct* eventPsi *process* P *and a transition* $P \overset{e}{\rightsquigarrow} P'$ *then* P' *is also syntactically correct.*

Proof. Having $P \overset{e}{\rightsquigarrow} P'$, we know from the transition rules of psi-calculi, and the syntactic restrictions of Definition 3.10 that $P = \mathbf{case}\ \varphi : \bar{e}\langle e\rangle.(\!|\{e\}|\!) \parallel Q$ where Q is syntactically correct and does not contain another (case or assertion process) P'_e, i.e., indexed by the same e. Recall that $\emptyset$ is the unit assertion $\mathbf{1}$, and that the minimal process $\mathbf{0}$ is equivalent with the $(\!|\emptyset|\!)$, which can be in place of Q so that the parallel composition ends. The transition thus is

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

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

Lemma 3.14 *(Correspondence configuration—frame). For any event structure* $\mathcal{E}$ *and configuration* C*, the frame of the* eventPsi*-process* $\textsc{espsi}(\mathcal{E}, C)$ *is the same as the configuration* C*.*

Proof. Denote $\textsc{espsi}(\mathcal{E}, C) = P_E^C$ defined as in Definition 3.9. The frame of P_E^C is the composition with $\otimes$ of the frames of P_e for $e \in E$. As P_e is either $(\!|\{e\}|\!)$ if $e \in C$ or $\mathbf{case}\ \varphi_e : \bar{e}\langle e\rangle.(\!|\{e\}|\!)$ then the frame of P_e would be either $\mathcal{F}((\!|\{e\}|\!)) = \{e\}$ or $\mathcal{F}(\mathbf{case}\ \varphi_e : \bar{e}\langle e\rangle.(\!|\{e\}|\!)) = \mathbf{1} = \emptyset$. Thus the frame of P_E is the union of $\emptyset$ and all events in C. □

Lemma 3.15 *(Transitions are preserved). For any event structure* $\mathcal{E}$ *and any of its configurations* C*, any transition from this configuration* $C \xrightarrow{e} C'$ *is matched by a transition* $\textsc{espsi}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} \textsc{espsi}(\mathcal{E}, C')$ *in the corresponding* eventPsi*-process.*

Proof. By Lemma 3.14 the frame of $\textsc{espsi}(\mathcal{E}, C)$ is the same as C. The assumption of the lemma, i.e., the existence of the step between configurations, implies that e is enabled by the configuration C. This means that $e \notin C$, which implies by Definition 3.9 that $\textsc{espsi}(\mathcal{E}, C) = \mathbf{case}\ \varphi_e : \bar{e}\langle e\rangle.(\!|\{e\}|\!) \parallel Q$. This implies that $\mathcal{F}(\textsc{espsi}(\mathcal{E}, C)) = \mathbf{1} \otimes \mathcal{F}(Q) = C$, with $e \notin C$, meaning that $\mathcal{F}(Q) = C$. Moreover, since C enables e it means that all $< e$ are in C and no $\sharp e$ is in C, which is the definition of entailment relation in eventPsi, i.e., $\mathcal{F}(\textsc{espsi}(\mathcal{E}, C)) \vdash \varphi_e$, which enables the step from $\textsc{espsi}(\mathcal{E}, C)$ that the lemma expects. After $\textsc{espsi}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} P'$ we have $P' = (\!|\{e\}|\!) \parallel Q$ and $\mathcal{F}(P') = \mathcal{F}((\!|\{e\}|\!)) \otimes \mathcal{F}(Q) = \{e\} \cup C = C'$. From the definition of the translation function espsi it is easy to see that $\textsc{espsi}(\mathcal{E}, C') = (\!|\{e\}|\!) \parallel Q$. □

Lemma 3.16 *(Transitions are reflected). For an event structure* $\mathcal{E}$ *and a configuration* C*, any transition* $\textsc{espsi}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} P'$ *is matched by a step* $C \xrightarrow{e} C'$*, with* $P' = \textsc{espsi}(\mathcal{E}, C')$*.*

Proof. We know that for $\textsc{espsi}(\mathcal{E}, C)$ to have a transition labelled with e it must be of the form $\textsc{espsi}(\mathcal{E}, C) = P_e \parallel Q$ where $P_e = \mathbf{case}\ \varphi_e : \bar{e}\langle e\rangle.(\!|\{e\}|\!)$, with $\varphi_e = (< e, \sharp e)$. We know from Lemma 3.14 that the frame of $\textsc{espsi}(\mathcal{E}, C)$ is the assertion corresponding to C, which is $\mathcal{F}(P_e \parallel Q) = \mathbf{1} \otimes \Psi_Q = \Psi_Q$. For the transition e to be enabled we also know from Definition 3.7 that $< e \subseteq \Psi_Q$ and $\sharp e \cap \Psi_Q = \emptyset$. From how φ_e is created in the Definition 3.9 we know that e must be enabled in $(\mathcal{E}, C)$. Therefore, we have the transition $(\mathcal{E}, C) \xrightarrow{e} (\mathcal{E}, C')$, where $C' = \{e\} \cup C$.

After a transition $P_e \parallel Q \overset{e}{\rightsquigarrow} (\!|\{e\}|\!) \parallel Q$ we have that the new frame of the process is $\{e\} \cup \Psi_Q$. From Definition 3.9 we see that $\textsc{espsi}(\mathcal{E}, C')$ would create an eventPsi-process where all but the sub-process for e will be the same as for $\textsc{espsi}(\mathcal{E}, C)$, and the sub-process P_e will be $(\!|\{e\}|\!)$ instead of $\mathbf{case}\ \varphi_e : \bar{e}\langle e\rangle.(\!|\{e\}|\!)$. This is the same process that we got after the transition in eventPsi. □

Theorem 3.17 *(Preserving interleaving diamonds). For an event structure* $\mathcal{E} = (E, \leq, \sharp)$ *with two concurrent events* $e||e'$*, then in the translation* $\textsc{espsi}(\mathcal{E}, \emptyset)$ *we find the behaviour forming the interleaving diamond, i.e., there exists a* C *s.t.*

$$\textsc{espsi}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} P_1 \overset{e'}{\rightsquigarrow} P_2 \quad \textit{and}$$
$$\textsc{espsi}(\mathcal{E}, C) \overset{e'}{\rightsquigarrow} P_3 \overset{e}{\rightsquigarrow} P_2.$$

Proof. In a prime event structure if two events e, e' are concurrent then there exists a configuration C reachable from the root which contains the conditions of both events, i.e., $< e \subseteq C$ and $< e' \subseteq C$, and does not contain any of the two events, i.e., $e, e' \notin C$. This can be seen from Remark 3.6(3) which ensures the existence of some configurations C_1, C_2, and $C_1 \cup C_2$, which contains both e, e'. Removing from this last configuration both e, e' we still obtain a configuration. Take this configuration as the one C sought in the theorem. Therefore we have the following steps in the event structure: $C \xrightarrow{e} C \cup \{e\}$, $C \xrightarrow{e'} C \cup \{e'\}$, $C \cup \{e\} \xrightarrow{e'} C \cup \{e, e'\}$, and $C \cup \{e'\} \xrightarrow{e} C \cup \{e, e'\}$.

Since C is reachable from the root then by Lemma 3.15 all the steps are preserved in the behaviour of the eventPsi-process $\text{ESPSI}(\mathcal{E}, \emptyset)$, meaning that $\text{ESPSI}(\mathcal{E}, C)$ is reachable from (i.e., part of the behaviour of) $\text{ESPSI}(\mathcal{E}, \emptyset)$.

Since $e, e' \notin C$ we have that $\text{ESPSI}(\mathcal{E}, C)$ is in the form $P_0 = P_e \parallel P_{e'} \parallel Q$ with P_e and $P_{e'}$ processes of kind **case**. From Lemma 3.14 we know that the frame of $\text{ESPSI}(\mathcal{E}, C)$ is the assertion corresponding to C, which is $\mathcal{F}(P_e \parallel P_{e'} \parallel Q) = \emptyset \cup \emptyset \cup \Psi_Q = \Psi_Q$.

From Lemma 3.15 we see the transitions between the eventPsi-processes: $\text{ESPSI}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} P_1 \overset{e'}{\rightsquigarrow} P_2$ with $P_2 = (\!|\{e\}|\!) \parallel (\!|\{e'\}|\!) \parallel Q$ as well as $\text{ESPSI}(\mathcal{E}, C) \overset{e'}{\rightsquigarrow} P_3 \overset{e}{\rightsquigarrow} P_4$ with $P_4 = (\!|\{e\}|\!) \parallel (\!|\{e'\}|\!) \parallel Q$. We thus have the expected interleaving diamond.

As an aside, remark that $\mathcal{F}(P_1) = \mathcal{F}(P_0) \otimes \{e\}$ and $\mathcal{F}(P_3) = \mathcal{F}(P_0) \otimes \{e'\}$ thus $\mathcal{F}(P_1) \otimes \mathcal{F}(P_3) = \mathcal{F}(P_0) \otimes \{e\} \otimes \{e'\} = \mathcal{F}(P_4)$, which says that $e \in \mathcal{F}(P_1) \wedge e' \notin \mathcal{F}(P_1) \wedge e' \in \mathcal{F}(P_3) \wedge e \notin \mathcal{F}(P_3) \wedge \mathcal{F}(P_1) \otimes \mathcal{F}(P_3) = \mathcal{F}(P_4)$. Using Lemma 3.14 these can be correlated with configurations and thus we can see the definition of concurrency from configurations as in Remark 3.6(3). □

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

Theorem 3.18 *(Reflecting interleaving diamonds). For any event structure $\mathcal{E}$, in the corresponding* eventPsi*-process* $\text{ESPSI}(\mathcal{E}, \emptyset)$*, for any interleaving diamond*

$$\text{ESPSI}(\mathcal{E}, C) \overset{e}{\rightsquigarrow} P_1 \overset{e'}{\rightsquigarrow} P_2 \quad \textit{and}$$
$$\text{ESPSI}(\mathcal{E}, C) \overset{e'}{\rightsquigarrow} P_3 \overset{e}{\rightsquigarrow} P_2$$

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

Proof. Since $\text{ESPSI}(\mathcal{E}, C)$ has two outgoing transitions labelled with the events e and e' it means that $\text{ESPSI}(\mathcal{E}, C)$ is in the form $P_0 = P_e \parallel P_{e'} \parallel Q$ with P_e and $P_{e'}$ processes of kind **case**. From Lemma 3.14 we know that the frame of $\text{ESPSI}(\mathcal{E}, C)$ is the assertion corresponding to C, which is $\mathcal{F}(P_e \parallel P_{e'} \parallel Q) = \emptyset \cup \emptyset \cup \mathcal{F}(Q) = \Psi_Q$.

We thus have that $e, e' \notin \Psi_Q$ and $P_0 \overset{e}{\rightsquigarrow} P_1$ and $P_0 \overset{e'}{\rightsquigarrow} P_3$. This means that for these two transitions to be possible it must be that the precondition for e and e' respectively must be met. Since $e, e' \notin \Psi_Q$ it must be that $e' \notin \pi_L(\varphi_e)$ and $e \notin \pi_L(\varphi_{e'})$. Since $\pi_L(\varphi_e)$ is the same as the set $< e$ and $\pi_L(\varphi'_e)$ the set $< e'$ we have the two parts of the Definition 3.3 that concern $\leq$ for the causal independence (concurrency) of the events e, e', i.e., $\neg(e' \leq e \vee e \leq e')$. After the two transitions are taken we have that $P_1 = (\!|\{e\}|\!) \parallel P_{e'} \parallel Q$ and $P_3 = P_e \parallel (\!|\{e'\}|\!) \parallel Q$. We thus have that $e \in \mathcal{F}(P_1)$ and $e' \in \mathcal{F}(P_3)$. For the transition $P_1 \overset{e'}{\rightsquigarrow} P_2$ to happen we must have that $e \notin \pi_R(\varphi_{e'})$ and for $P_3 \overset{e}{\rightsquigarrow} P_4$ 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 3.3 concerning the conflict relation, i.e., $\neg(e' \sharp e)$. This completes the proof, showing $e||e'$. □

Theorem 3.19 *(Syntactic restrictions). For any syntactically correct* eventPsi*-process P_{ES} there exists an event structure $\mathcal{E}$ and a configuration $C \in \mathbb{C}_{\mathcal{E}}$ s.t.*

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

Proof. From an eventPsi-process P_{ES} defined according to the syntactic restrictions of Definition 3.10, we show how to construct an event structure $\mathcal{E} = (E, \leq, \sharp)$ and a configuration C. We have that P_{ES} is built up of assertion processes and **case** guarded outputs, i.e.,

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

Because of the third restriction on P_{ES} (i.e., from Definition 3.10(3)) we know that E_c and E_r are sets, since P_{ES} cannot have in parallel two assertion processes with the same assertion, nor two **case** processes with the same channel. Moreover, these two sets are disjoint.

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

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

Notation. For a syntactically correct process P (i.e., restricted according to Definition 3.10) we will use the notation E_P for the events associated to the process P as defined in the above proof, i.e., $E_P = E_c \cup E_r$.

3.3. Action refinement

Below we show how to refine eventPsi processes corresponding to *action refinement* for labelled event structures [17] as recalled below. The intuition of action refinement is to be able to give actions (which are thought of as possible abstractions) more structure, by replacing every event labelled by a particular action with a finite, conflict free event structure (with events possibly labelled by other actions). For example one action can be refined into a sequence of actions, or in general any deterministic finite concurrent process.

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

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

Definition 3.20 *(Refinement for prime event structures).* For an event structure $\mathcal{E}$ with events labelled by $l : E \to Act$ a function $ref : Act \to \mathbb{E}_{\not\sharp}$ is called a *refinement function* (for prime event structures) iff $\forall a \in Act : ref(a)$ is a non-empty, finite and conflict-free labelled prime event structure.

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

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

Example 3.21. Fig. 4(a) provides an illustrative example (taken from [17]) of action refinement applied to a process which receives and sends data. We may think of giving more details to the sending event by refining it with a sequential process which first prepares the data and then carries out the actual sending. This refined process is shown in Fig. 4(b). In turn, Fig. 4(c) shows a further refinement, where the preparation of data is refined by a process which in parallel formats the data and asks permission to send.

Remark 3.22. Action refinement as presented here was introduced by Wirth [16] under the name of *stepwise refinement* and is quite different than the more recent notion of refinement in process algebras where the refined process is seen as an *implementation* of the abstract one. In these settings usually refinement is seen as an inclusion-like relation of the behaviours, such as trace inclusion or simulation.

We can define a refinement function $ref^{\Psi} : Act \to \mathbf{P}_{\text{eventPsi}}$ for eventPsi-processes from a given refinement function ref for event structures as follows: $ref^{\Psi}(a) = \textsc{espsi}(ref(a))$. For syntactically correct eventPsi-processes these two functions are one-to-one inter-definable in the sense that for eventPsi-processes that represent finite conflict-free event structures (i.e., that have the right elements of the conditions always empty) we can define ref from ref^{Ψ} by an analogous definition as before, going through Theorem 3.19. In the rest of this section, we prefer to work with the simpler notation provided by ref.

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

Definition 3.23. Given a refinement function ref for event structures over events E, we define an operation $\mathbf{ref}^{\Psi}$ that refines an eventPsi-process over events E to a new one over the terms

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

A syntactically correct eventPsi-process P, built according to Definition 3.10, with frame $\mathcal{F}(P) = \Psi_P$, is refined into a process

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

and

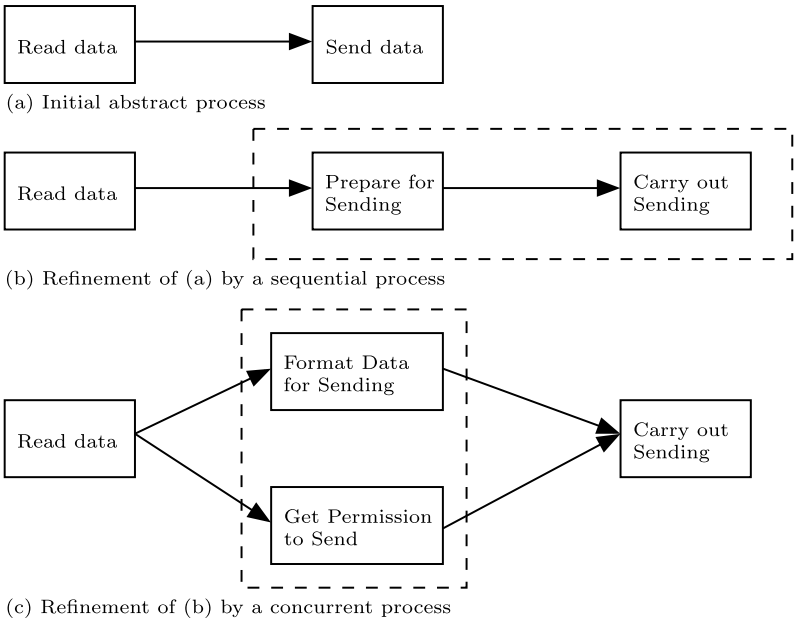

Fig. 4. Example for action refinement from [17].

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

with the conditions being

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

where

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

and

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

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

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

Theorem 3.24 *(Refinement in* eventPsi *corresponds to that in ES). For any prime event structure $\mathcal{E}$ we have that:*

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

Proof. As $\mathbf{T} = E$ and $\mathbf{T}^{\Psi}$ is built from $\mathbf{T}$ in Definition 3.23 with rules analogous to those E_{ref} is built from E in Definition 3.20, we have that $\mathbf{T}^{\Psi} = E_{ref}$. Since the processes we work with are parallel compositions of assertion and **case** processes, it means we have to show that any assertion processes on the left is also found on the right of the equality (and vice versa), and the same for the **case** processes. Since we work with the empty initial configuration, then there are no assertion processes on either sides.

The **case** processes on the left side of the equality are those generated by ESPSI from the pairs of events returned by the **ref** from the event structure $\mathcal{E}$, i.e., $P_{(e,e')} = \textbf{case}\ \varphi_{(e,e')} : \overline{(e,e')}\langle(e,e')\rangle.(\!|\{(e,e')\}|\!)$ with the condition $\varphi_{(e,e')} = (< (e,e'), \sharp(e,e'))$ which is build according to Definition 3.20 from $\sharp_{\mathcal{E}} e$, $<_{\mathcal{E}} e$, and $<_{ref(l(d))} e'$. On the right side we have **case** processes for the original process before the refinement, with their respective conditions. But the $\mathbf{ref}^{\Psi}$ replaces each

one of these P_e with several **case** processes, one for each new pair (e, e') that involves e. Therefore, according to Definition 3.23 we will have $P_{(e,e')} = \textbf{case}\ \varphi_{(e,e')} : \overline{(e, e')}\langle(e, e')\rangle.(\!|\{(e, e')\}|\!)$ with its condition being built from $\pi_R(\varphi_e)$, $\pi_L(\varphi_e)$, and $<_{ref(l(d))} e'$. These are respectively the same as the ones used on the left side. Checking that any case process from the right side is found in the left side is done similarly. □

In this section we gave a representation of an encoding of the prime event structures into an instance of psi-calculi which preserves the causality relation and thereby also the notion of action refinement. To do this, we made special use of the logic of psi-calculi, i.e., of the assertions and conditions and the entailment between these, as well as the assertion processes. It is noteworthy that we have not used neither names nor the communication mechanism of psi-calculi, which is known to increase expressiveness.[8]

4. DCR graphs as psi-calculi

4.1. Background on DCR graphs

Dynamic Condition Response graphs (DCR graphs) [15,45] is a model of concurrency which generalises event structures in two dimensions: Firstly, it allows finite models of (regular) infinite behaviour, while retaining the possibility of infinite models. The finite models are regular in the automata-theoretic sense, i.e. they (if concurrency is ignored) capture exactly the languages that are the union of a regular and an omega-regular language [46]. Finite DCR graphs have found applications in practice for the description, implementation and automated verification of flexible workflow systems [47,18]. Infinite DCR graphs allow for representation of non-regular behaviour and denotational semantics. Secondly, the DCR graphs model provides an event-based notion of acceptance criteria for both finite and infinite computations in terms of scheduled responses. In the present paper we focus on the first dimension, leaving the interesting question of representing event-based acceptance criteria for infinite computations to future work. We follow the notations for DCR graphs from [15,45].

Definition 4.1 *(DCR graphs).* We define a *Dynamic Condition Response Graph* to be a tuple $\mathcal{D} = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ where

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

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

A *marking* $M = (Ex, Re, In)$ represents a state of the DCR graph. One should understand Ex as the set of *executed* events, Re the set of scheduled *response* events[9] that must happen sometime in the future or become excluded for the run to be accepting (see Definition 4.4), and In the set of currently *included* events. The five relations impose constraints on the events and dictate the dynamic inclusion and exclusion of events.

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

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

Definition 4.2 *(Enabling events).* For a DCR graph $\mathcal{D} = (E, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%)$ with an initial marking $M = (Ex, Re, In)$, we say that an *event $e \in E$ is enabled in M*, written $M \vdash e$, iff

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

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

[8] In π-calculus communication of channel names allows to reach Turing completeness [43], in contrast to CCS with only synchronisation and replication (called $\text{CCS}_!$) where the expressiveness is weaker [44].

[9] Similar to the notion of restless events in [48, ch. 6.4.].

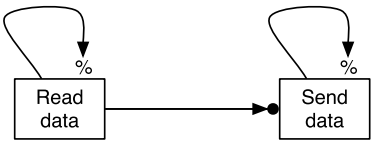

Fig. 5. Simple DCR graph of a read-send process.

Definition 4.3 *(Transitions).* The behaviour of a DCR graph is given through *transitions between markings* done by executing enabled events. The result of the execution in a DCR graph $\mathcal{D} = (E, M_0, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%)$ from marking $M = (Ex, Re, In)$ of an enabled event $M \vdash e$ results in the new marking

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

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

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

Definition 4.4 *(Accepting runs [15]).* A *run* of a DCR graph with initial marking M_0 is a (possibly infinite) sequence of transitions $M_i \xrightarrow{e_i} M_{i+1}$, with $0 \leq i < k$, $k \in \mathbb{N} \cup \{\omega\}$, and $M_i = (Ex_i, Re_i, In_i)$. A run of a DCR graph is *accepting* (or completed) if it holds that

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

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

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

It is worth noting that the labelling function on events adds the possibility of non-determinism by taking the language of a DCR graph to be the sequences of labels of events (abstracting from the events) of accepting runs. This extra level of labelling increases the expressive power of finite DCR graphs, which have been shown to capture exactly the languages that are the union of a regular and an omega-regular language [46]. In contrast, by following an encoding along the lines of [49] unlabelled, finite DCR graphs can be represented by Linear-time Temporal Logic (LTL), which is known to be strictly less expressible than omega-regular languages.

As given by the mapping in Definition 4.5 below, prime event structure can be seen as a special case of a DCR graph (see [15, Prop. 1&3] for details) where the exclusion relation (capturing the conflict relation) is reflexive and symmetric, the condition relation (capturing the causality relation) is irreflexive and transitive, and the include, response, and milestone relations are empty. The initial marking has no executed events, no scheduled responses and all events are included. Hereto comes of course the two conditions on the causality and conflict relation of event structures, i.e. finite causes and hereditary conflict. The reflexivity of the exclusion relation and emptiness of the inclusion relation imply that events can be executed at most once.

Definition 4.5 *(Prime event structures as DCR graphs [15]).* Define a mapping **dcr** which takes an event structure $\mathcal{E} = (E, \leq, \sharp, l)$ and returns its presentation as a DCR graph $(E, M, <, \emptyset, \emptyset, \emptyset, \sharp \cup \{(e, e) \mid e \in E\})$ with the marking $M = (\emptyset, \emptyset, E)$.

Example 4.6. Consider the small DCR graph $\mathcal{D}$ shown in Fig. 5, corresponding to the event structure of Fig. 4(a) which was representing a process of first reading data (R) and then sending data (S). The DCR graph is formalised as

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

Here we can see that each event removes itself from the included set when it happens, and that for S to happen its prerequisite R must happen. This corresponds to the causality relation in the event structure in Fig. 4(a).

In DCR graphs is possible to demand that if we read some data we will eventually send this data. This is modelled in the DCR graph of Fig. 6 formalised as

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

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

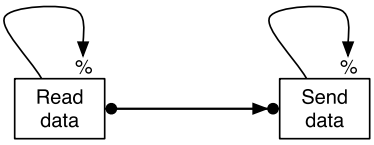

Fig. 6. DCR graph with added responses.

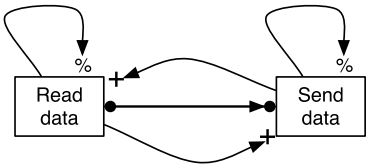

Fig. 7. Message forwarder DCR graph with possible infinite execution.

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

Example 4.7. A message forwarding machine, where the events can happen several times, but alternating, can be represented by the DCR graph in Fig. 7 formalised as

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

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

4.2. Encoding DCR graphs

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

However, in contrast to the encoding of event structures, it appears that we need the communication constructs on processes to keep track of the *current marking* of a DCR graph. The expressiveness of DCR graphs seems not to allow for a simple way of updating the marking, as was the case for event structures where union with the newly executed event was enough. But once we use the communication we get a nice natural encoding for DCR graphs in a psi-calculus instance. The idea is to internally communicate a term representing the current marking, and incorporate a generation (or age) of an assertion and then keep the assertion with the latest generation when composing assertions. Generations are inspired by [50], where they are used in a similar way to represent the changing topology of a mobile communication network.

Since a step is now an internal communication, we can no longer observe the event that happened by just looking at the communication channel. Instead, we record all executed events in a multi-set part of the assertion. The underlying set of this multi-set corresponds to the set of executed events in the marking of the DCR graph. By recording also the multiplicity of each event one can identify, by multi-set subtraction, which event happened in a transition, as expressed formally below.

Notation 4.8. For a multi-set S we will denote its underlying set by $\lfloor S \rfloor$ and write $|S|$ for the number of elements of S, summing multiplicities.

Definition 4.9 *(*dcrPsi *instance).* Given a set of *constants* E (denoting events), we define the dcrPsi instance over a set of names $\mathcal{N}$ as:

$$\mathbf{T} \stackrel{def}{=} \mathcal{N} \cup \mathbf{A} \cup \mathbb{N}_{op}^{E} \cup 2_{op}^{E} \cup \mathbb{N}_{op} \qquad \mathbf{A} \stackrel{def}{=} (\mathbb{N}_{op}^{E} \times 2_{op}^{E} \times 2_{op}^{E} \times \mathbb{N}_{op}) \cup \mathbf{1} \qquad \mathbf{1} \stackrel{def}{=} \bot$$

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

where $\mathbb{N}_{op}^{E}$ are the multi-sets over E with the operators of Example 2.19 and possibly containing names from $\mathcal{N}$, $\mathbb{N}_{op}$ is the data structure capturing natural numbers from Example 2.18, and $\bot$ is a special constant term assertion. (We will refer

to assertions containing only ground terms as ground assertions, and assertions containing names as open assertions.) For ground assertions, i.e., when Ex, Re, In, g denote ground terms, define

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

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

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

Entailment $\vdash$ is defined as:

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

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

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

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

Notation 4.10. We denote ground assertions, and the respective four ground terms, by (Ex, Re, In, g). In the few cases where we need to talk about open terms we will use capital letters X_E, X_R, X_I, X_G possibly indexed to indicate the respective component in the tuple, to stand for a name. We do not always mention explicitly if the assertions or terms are ground when this is clear from the context or the notation.

Terms can be either a name or assertions (and their components) which will be the data communicated. Assertions are four-tuples of one multi-set and two sets containing events, whereas the fourth element is a number which we intend to hold the *generation* of the assertion.[10] The multi-sets capture which events have been executed, and also count how many times each event has been executed. This will allow, in Definition 4.23, to infer from the change in the frame, which event happened in an execution step. The second set holds those events that are pending responses, and the third set those events that are included. The multi-set and set operations in the terms allow us to construct terms for updating the sets when an event is executed. The underlying set of the multi-set represents the first set of a marking from DCRs, whereas the two other sets of an assertion represent the second and third set of a marking of a DCR graph. The generation number helps to get the properties of the assertion composition, which are somewhat symmetric, but still have the composition return only the latest marking/assertion (i.e., somewhat asymmetric).

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

When any of the assertions contain names (i.e., either one of the tuple elements is a name or a term contains a name) then the composition returns the ground assertion when it exists or the identity assertion, when both assertions contain names. In other words, we are interested only in the ground assertions, those containing the actual data, without undefined parts. This makes the composition operation associative, commutative, compositional wrt. assertion equivalence,[11] and with identity being the special constant term $\perp$.

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

As in [50], we added the set of natural numbers as conditions for technical reasons, so that assertion equivalence will be compositional. Without this we would have $(Ex, Re, In, 0) \simeq (Ex, Re, In, 2)$, but when composed with another assertion $(Ex', Re', In', 1)$, where $In \cap In' = \emptyset$, we would get $(Ex, Re, In, 0) \otimes (Ex', Re', In', 1) \not\simeq (Ex, Re, In, 2) \otimes (Ex', Re', In', 1)$ which contradicts compositionality: On the left side we would keep the assertion with highest generation 1 whereas on the right side we would keep the one with 2. As the sets of *included* events are different we have that they cannot entail the same conditions. With the natural numbers as conditions we can distinguish between the two assertions $(Ex, Re, In, 0) \not\simeq (Ex, Re, In, 2)$ since we have that $(Ex, Re, In, 0) \nvdash 2$ but $(Ex, Re, In, 2) \vdash 2$.

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

[11] Recalling from Section 2, compositionality refers to: $\Psi \simeq \Psi' \Rightarrow \Psi'' \otimes \Psi \simeq \Psi'' \otimes \Psi'$.

Remark 4.11 *(On sorting).* We rely on sorting to properly define in dcrPsi the terms, substitutions, matching, etc. We rarely mention sorting aspects, only when necessary for clarifications, and rely on the intuition in most cases. For instance, in Definition 4.9 we silently use sorts for several aspects: to distinguish the different kinds of terms (like a sort for natural numbers, another sort for multi-sets); several corresponding name sorts; the assertion tuple operand is multi-sorted; the substitution takes care that names on the respective place in a tuple are replaced by terms of respective sort. We thus make careful use of notation to be in accordance with the sorting aspects; e.g., the notation (a, a, a, b) is unacceptable, opposed to (a, c, c, b) which allows both second and third elements of a tuple to be the same term.

Lemma 4.12. *For two assertions* $\Psi = (Ex, Re, In, g), \Psi' = (Ex', Re', In', g')$ *we have that* $\Psi \simeq \Psi' \Rightarrow g = g'$.

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

Lemma 4.13 *(Correctness of* dcrPsi*). The* dcrPsi *instance fulfils the requirements of being a psi-calculi instance, cf. Definitions 2.2 and 2.4.*

Proof. We have to show that the channel equivalence and the composition of assertions conform with the requirements from Definition 2.4. We also need to show that the operators and nominal datatypes of Definition 2.2 are well defined.

For the channel equivalence it is easy to see that symmetry and transitivity are respected since in Definition 4.9 (last line) channel equivalence is defined in terms of name equality.

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

We first point out that all open assertions are assertion equivalent through the fact that they cannot entail any conditions (i.e., entailment in Definition 4.9 is undefined for open assertions). The same goes for **1** which also does not entail any conditions, and thus assertion equivalent with all open assertions. Whenever we compose two open assertions we obtain **1**, i.e., composition of open assertions results in the identity assertion that is assertion equivalent with all open assertions. (From here on we will refer to **1** as an open assertion for the sake of simplicity because we only care about assertion equivalence for the correctness proofs). Since the four requirements from Definition 4.9 on assertion composition are defined in terms of assertion equivalence, then they are trivially satisfied when only open assertions are involved.[12]

For the case where we have a mix of open and ground assertions, the composition returns the ground assertion (i.e., open assertions are absorbed by any ground ones). It is easy to check the four requirements from Definition 4.9. Also note that it is not possible for any ground assertion to be equivalent with any open assertion since the ground assertion will at least entail the condition g when is its generation.

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

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

For compositionality we know, by Lemma 4.12, that if two assertions Ψ, Ψ' are equivalent then they have the same generation g. Therefore, when composed with another assertion Ψ'' with generation g'' we must treat three cases. If $g < g''$ then for both $\Psi \otimes \Psi''$ and $\Psi' \otimes \Psi''$ we obtain Ψ''. If $g = g''$ then on both sides we obtain the assertion $(\emptyset, \emptyset, \emptyset, g)$. When $g > g''$ we have that $\Psi \otimes \Psi'' = \Psi$ and $\Psi' \otimes \Psi'' = \Psi'$ which are equivalent by assumption.

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

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

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

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

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

Lemma 4.14 *(Correlation between entailment in DCRs and* dcrPsi*). For any DCR graph $\mathcal{D}$ we have that*

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

Proof. Follows directly from Definition 4.2 and Definition 4.9 □

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

Definition 4.15 *(DCR-graphs to* dcrPsi*).* We define a function $\textsc{dcrpsi}(\mathcal{D}, Hi)$ which takes a DCR $\mathcal{D} = (E, M{\rightarrow\bullet}, {\bullet\!\!\rightarrow}, {\rightarrow\diamond}, {\rightarrow\!\!+}, {\rightarrow\%}, L, l)$, with initial marking $M = (Ex, Re, In)$, and a multi-set Hi representing the history of events that have happened, with underlying set $\lfloor Hi \rfloor = Ex$, and returns a dcrPsi process

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

where

$$P_k = (\!|(Hi, Re, In, |Hi|)|\!) \parallel \overline{m}\langle(Hi, Re, In, |Hi|)\rangle.\mathbf{0} \qquad \text{and} \qquad P_E = \prod_{e \in E} P_e$$

with

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

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

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

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

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

Notation 4.16. In the context of Definition 4.15, i.e., when encoding a DCR through the $\textsc{dcrpsi}$, we will use the following shorthand notation to stand for the often and similar way of updating a marking:

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

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

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

would produce the term

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

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

Example 4.17. Taking the DCR graph $\mathcal{D}_1$ from Fig. 5, we can create a dcrPsi-process $P = \text{DCRPSI}(\mathcal{D}_1, \emptyset)$, with initial generation $|\emptyset| = 0$, as follows

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

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

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

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

Definition 4.18 *(Syntactic restrictions for* dcrPsi*).* We define a dcrPsi-process P to be *syntactically correct* for a set of events E if it is of the following form, up to structural congruence:

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

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

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

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

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

Lemma 4.19. *In a syntactically correct* dcrPsi*-process P we have that:*

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

In the following when we refer to a dcrPsi-process we will assume that it is a syntactically correct process, over some finite set E of event constants. In Theorem 4.28 we prove that for any syntactically correct P there exists a DCR graph which is bisimilar to P. Before that we need a few preparatory results.

Lemma 4.20. *For any $\mathcal{D}$ and Hi, the* dcrPsi*-process* DCRPSI$(\mathcal{D}, Hi)$*, if defined, is syntactically correct.*

Proof. Follows directly from Definition 4.15 of DCRPSI. □

Lemma 4.21 *(Frame-marking correspondence). For any $\mathcal{D}$ and Hi, then the following statements are equivalent*

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

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

Lemma 4.22 *(Transitions preserve syntactic correctness). For any syntactically correct* dcrPsi*-process*

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

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

1. $\alpha = \tau$
2. $$P_1 \equiv (\nu m)((\prod_{k\le g\le g_0+1} (\!|\Psi_g|\!)) \parallel \overline{m}\langle(Ex_1, Re_1, In_1, g_1)\rangle.\mathbf{0} \parallel (\prod_{e\in E} P_e))$$
3. $\exists e \in E$ *such that:* $Ex_1 = Ex_0 + \{e\}$, $Re_1 = (Re_0 \setminus \{e\}) \cup e\bullet\!\!\rightarrow$, $In_1 = (In_0 \setminus e\!\rightarrow\!\%) \cup e\!\rightarrow\!+$, $g_1 = s(g_0)$.
In particular, it follows that transitions of syntactically correct dcrPsi*-processes preserve syntactic correctness.*

Proof.

1. In psi-calculi there are three different types of transitions: input, output, and τ transitions. Syntactically correct dcrPsi-processes communicate over only one channel name m which is guarded, and therefore there cannot exist input nor output transitions. Thus the only possibility is τ-transitions.
2. To see that P_1 is syntactically correct note that any transition in P_0 is between the unique top level output $\overline{m}\langle(Ex_0, Re_0, In_0, g_0)\rangle.\mathbf{0}$ and a **case** -guarded input

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

   coming from the replication of some P_e. After the transition and substitution updating the marking, the output becomes $\mathbf{0}$, whereas the **case** process becomes $\overline{m}\langle(Ex_1, Re_1, In_1, g_1)\rangle.\mathbf{0} \parallel (\!|(Ex_1, Re_1, In_1, g_1)|\!)$ where Ex_1, Re_1, In_1, g_1 are as in the statement, cf. Definition 4.15 and Notation 4.16. Since $Ex_1 = Ex_0 + \{e\}$ we can find the event e responsible for the current τ-transition through $Ex_1 \setminus Ex_0 = \{e\}$. In consequence:
   (a) The single output of P_0 has been reduced in P_1 to $\mathbf{0}$ and we obtained exactly one new output of the correct form, as required.
   (b) Since no assertions from P_0 were removed, we can write the assertions in P_1 as $\prod_{k\le g\le g_0} (\!|\Psi_g|\!) \parallel (\!|(Ex_1, Re_1, In_1, s(g_0))|\!)$, which can be written as $\prod_{k\le g\le s(g_0)} (\!|\Psi_g|\!)$.
   (c) The application of rule (REP) reduces P_e to itself in parallel with the **case**-process that participated in the communication above. Therefore, the product of P_e processes remains the same, whereas the case process becomes the new output and a new assertion process.
3. Follows directly from the Notation 4.16 for U_e after substitutions. □

Based on Lemma 4.22 we can define transitions labelled by event constants between two dcrPsi-processes as follows.

Definition 4.23 *(Event transitions).* Define $P \overset{e}{\rightsquigarrow} P'$ iff $\mathbf{1} \rhd P \xrightarrow{\tau} P'$ and $Ex' \setminus Ex = \{e\}$ from $\mathcal{F}(P) = (Ex, Re, In, g)$ and $\mathcal{F}(P') = (Ex', Re', In', g')$. Define *event-labelled transition systems*, denoted $\mathcal{LTS}$, to have as states all dcrPsi-processes and transitions between states defined by the above event-labelled transitions. For some dcrPsi-process P, we call the event-labelled transition system of P, denoted $\mathcal{LTS}(P)$, only the part of $\mathcal{LTS}$ reachable from P.

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

Proposition 4.24 *(Preserving behaviour). For*

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

then

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

s.t.

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

Proof. First it is easy to see that $\mathcal{F}(P) = \mathcal{F}(\text{DCRPSI}(\mathcal{D}, Hi))$ from Definitions 4.15 and 4.18.

From Definitions 4.2, 4.9 and 4.15, and Lemma 4.21, we can see that an event is enabled in P iff it is enabled in $\mathcal{D}$. Therefore, whenever a transition exists on one side of the double implication, it exists on the other side as well.

It is easy to see from Definitions 4.3 and 4.15, that after a transition $P \xrightarrow{e} P'$, the updates to the frame $\mathcal{F}(P) = (Ex, Re, In, g)$ that we see in the frame of $\mathcal{F}(P')$ are the same as when updating the marking to $M' = (Ex', Re', In')$ in the transition $\mathcal{D} \xrightarrow{e} \mathcal{D}'$, with the exception of Ex being a multi-set and the generation g. In particular, $Ex' = Ex + \{e\}$, which in a marking it means that e is added to Ex if it was not already there, whereas in the assertion the multiplicity of e is increased.

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

Definition 4.25 *(Event bisimulation).* We define event bisimulation $\overset{\bullet}{\sim}$ between dcrPsi-processes $P \overset{\bullet}{\sim} Q$, to be the standard bisimulation between their corresponding event-labelled transition systems $\mathcal{LTS}(P) \sim \mathcal{LTS}(Q)$. This particularly means that there exists a relation $R \subseteq \mathcal{LTS}(P) \times \mathcal{LTS}(Q)$ between the states of the two transition systems such that $(P, Q) \in R$ and for any e

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

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

Theorem 4.26. *For a DCR graph $\mathcal{D}$ with marking $M = (Ex, Re, In)$ then*

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

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

We show that the following set is a bisimulation:

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

This is equivalent to the single steps coinductive statement of Definition 4.25. The initial pair, for the empty sequence, is $(\mathcal{D}, \text{DCRPSI}(\mathcal{D}, Ex))$.

Proving that all pairs respect the requirements of Definition 4.25 is easy by induction over the length k. The induction has as basis the above initial pair. The Proposition 4.24 ensures that whenever a transition exists from one element of the pair, then the same transition exists from the other element. By the above construction, this new pair is in our bisimulation relation, thus respecting Definition 4.25. □

Proposition 4.27 *(Determinism in* dcrPsi*). Syntactically correct* dcrPsi*-processes are deterministic; i.e., in the execution graph, at any point P if $P \overset{e}{\rightsquigarrow} P'$ then P' is unique.*

Proof. For any syntactically correct dcrPsi-processes we know that there is only one output process, cf. Lemma 4.19(3), and all input processes have a unique event e, cf. Lemma 4.19(2). Since event-labelled transitions are communications between one input and the single output, cf. Definition 4.23, then there can be at most one transition labelled by e. Lemma 4.22 ensures that any point in the execution graph is a syntactically correct process, thus finishing the proof. □

Theorem 4.28 *(Syntactic restrictions). For any syntactically correct* dcrPsi*-process P_{DCR}, i.e., built according to the syntactic restrictions in Definition 4.18, there exists a DCR graph $\mathcal{D}$ s.t.*

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

Proof. We take the set of events of the DCR graph to be the set of event constants of the dcrPsi instance. We know from Definition 4.18 that P_{DCR} is built as

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

where P_e is of the form

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

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

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

The bisimulation follows easily from Theorem 4.26. □

4.3. *Correlation between* dcrPsi *and* eventPsi

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

$$\text{are } \textsc{espsi}(\mathcal{E}) \text{ and } \textsc{dcrpsi}(\mathbf{dcr}(\mathcal{E})) \text{ similar in any way?}$$

Answering this question is not immediate. First of all, ESPSI translates into the eventPsi instance, whereas DCRPSI into the dcrPsi instance, and these two instances work with different terms and operator definitions. Even more, the encoding of event structures exhibits behaviour through *labelled transitions*, whereas for the encoding of DCRs we need to look into the frames of the processes before and after a transition to find the event that made this transition (cf. Lemma 4.22). Therefore, to find the correspondence that we are looking for we need to work in the same psi-instance, or establish correlations between the instances.

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

Lemma 4.29. *For an event structure $\mathcal{E}$ the* dcrPsi*-process* $\textsc{dcrpsi}(\mathbf{dcr}(\mathcal{E}))$ *has:*

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

Proof. From Definition 4.5 we know that the DCR graph $\mathbf{dcr}(\mathcal{E})$ has $\rightarrow\diamond = \emptyset$, which implies that the encoding function DCRPSI produces conditions $(Co, \emptyset, e)$ with the second element always empty. The initial assertion is made directly from the initial marking in the start process $P_{|Hi|}$ in Definition 4.15, which in the case of $\mathbf{dcr}(\mathcal{E})$ is $(\emptyset, \emptyset, E)$, cf. Definition 4.5. □

Lemma 4.30. *If* $\textsc{dcrpsi}(\mathbf{dcr}(\mathcal{E}), Ex) \rightsquigarrow^* P \overset{e}{\rightsquigarrow} P'$ *and frame $\mathcal{F}(P) = (Ex, \emptyset, In, g)$, then the new frame is*

$$\mathcal{F}(P') = (Ex{+}\{e\}, \emptyset, In \setminus (\sharp e \cup \{e\}), s(g)).$$

Proof. The fact that the generation increases to $s(g)$ is easy to see from how the assertions are created on transitions. The proof of Lemma 4.22 shows that the first element of the updated frame will be $Ex' = Ex{+}\{e\}$. From Definition 4.5 of $\mathbf{dcr}(\mathcal{E})$ we have $\rightarrow\!\!\% = \sharp \cup \{(e, e) \mid e \in E\}$ which implies that $\rightarrow\!\!\%e = (\sharp e \cup \{e\})$. Since $\rightarrow\!\!+ = \emptyset$, we have that the update of the third element of an assertion, when event e happens, is $In' = (In \setminus e\rightarrow\!\!\%) \cup e\rightarrow\!\!+ = (In \setminus (\sharp e \cup \{e\})) \cup \emptyset = In \setminus (\sharp e \cup \{e\})$. From Definition 4.5 we also have $\bullet\!\!\rightarrow = \emptyset$, which implies that the second element of the new frame is $Re' = (\emptyset \setminus \{e\}) \cup \emptyset = \emptyset$. □

Lemma 4.31. *For a* dcrPsi*-process* $\textsc{dcrpsi}(\mathbf{dcr}(\mathcal{E}), Ex) \rightsquigarrow^* P$, *if $\mathcal{F}(P) = (Ex, Re, In, g)$ then*

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

Proof. For each of the three points of the statement we use an inductive argument. For 1 the base case is given by Lemma 4.29(2) which shows that in the execution of the process initially we have $Re = \emptyset$. For the inductive case we use Lemma 4.30.

For 2 the base case is given by Lemma 4.29 which says that initially $Ex \cap In$ is the same as $\emptyset \cap E = \emptyset$. For the inductive case assume that the frame is $\mathcal{F}(P) = (Ex, \emptyset, In, g)$, with $Ex \cap In = \emptyset$ from the induction hypothesis. Consider a transition $P \overset{e}{\rightsquigarrow} P'$ and we show that the second claim holds for P'. Having the transition implies that the event e is enabled in the frame of P; i.e., Definition 4.23 implies a communication with a **case** process P_e for which the φ_e is enabled by $\mathcal{F}(P)$, which by Definition 4.9 implies that $e \in In$. From Lemma 4.30 we know that $\mathcal{F}(P') = (Ex' = Ex{+}\{e\}, Re' = \emptyset, In' = In \setminus (\sharp e \cup \{e\}), s(g))$ which only adds $\{e\}$ to Ex' compared to Ex. Since $e \in In$ it means that $Ex' \cap In = \{e\}$. But for In' we remove e from In, thus we have that $Ex' \cap In' = \emptyset$.

For 3 the base case is given by Lemma 4.29 which says that initially $In = E$ and $Ex = \emptyset$, thus allowing for the equality $In = E \setminus (\cup_{e \in \emptyset})(\sharp e \cup \{e\})$. For the induction case we assume that we have a process P with frame $(Ex, \emptyset, In, g)$ where

$In = E \setminus (\cup_{e \in Ex}(\sharp e \cup \{e\}))$. Consider a transition $P \overset{f}{\rightsquigarrow} P'$, implying by definition that $f \in In$, and the new frame $\mathcal{F}(P') = (Ex' = Ex + \{f\}, \emptyset, In', s(g))$ which has $In' = In \setminus (\sharp f \cup \{f\})$ by Lemma 4.30. Using the induction hypothesis, this is equal to $E \setminus (\cup_{e \in Ex}(\sharp e \cup \{e\})) \setminus (\sharp f \cup \{f\}) = E \setminus (\cup_{e \in Ex'}(\sharp e \cup \{e\}))$. □

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

Definition 4.32 *(Correlations between assertions).* For Ψ_C an assertion of eventPsi and $g \in \mathbb{N}$ a natural number define

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

We used the notation Ψ_C to remind that the assertion in eventPsi is a set of events corresponding to a configuration C in the event structure, cf. Lemma 3.14. The above definition of embedding is motivated by Lemma 4.31, in particular, having the second element of the assertion being $\emptyset$ comes from Lemma 4.31(2) and the shape of the third element is because of Lemma 4.31(3).

The opposite direction of the embedding from Definition 4.32 is obvious the projection on the first element of the dcrPsi assertion, keeping only the support set of the multi-set; denote this projection by $\mathbf{prj}(\Psi)$.

From Lemma 4.29 we can define a similar embedding of conditions, which we will also denote by $\mathbf{emb}^e$. We use these embeddings in the proofs and definitions later on.

Definition 4.33 *(Correlations between conditions).* For some event constant e and condition $(S_<, S_\sharp)$ of eventPsi define

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

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

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

Lemma 4.34 *(Correlation between entailment relations). For an event structure $\mathcal{E}$ and its encoding* ESPSI($\mathcal{E}$)*, and one of its conditions $\varphi_e = (< e, \sharp e)$ entailed by some assertion Ψ, then*

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

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

Proof. We need to prove that

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

under the assumptions that $< e \subseteq \Psi$ and $\sharp e \cap \Psi = \emptyset$, coming from the definition of entailment for eventPsi. For proving the above, the definition of entailment for dcrPsi requires that we prove three points (see Definition 4.9)

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

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

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

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

Lemma 4.35. *No event in a process* DCRPSI(**dcr**($\mathcal{E}$)) *can happen more than once.*

Proof. For an event to be enabled the corresponding condition must be enabled by the assertion and the frame of the process, which from the definition implies that $e \in In$. We also know that initially there are no events that have happened and $Ex = \emptyset$. From Lemma 4.30 we know that after a transition the respective event is added to the new Ex and from Lemma 4.31 we know that any events in Ex cannot simultaneously be in In. Since Ex does not decrease we thus have that if an event e has happened it will never be enabled again. □

We are now ready to define an embedding of eventPsi processes in the context of Theorem 3.19, i.e., generated by ESPSI, into dcrPsi-processes. Assertions in eventPsi are sets and in the next definition we use the number of elements of the set Ψ as the g parameter that the $\mathbf{emb}^g$ requires when applied over assertions.

Definition 4.36 *(Embedding* eventPsi *into* dcrPsi*).* We define an embedding function **emb** which takes an eventPsi-processes ESPSI($\mathcal{E}$) generated from an event structure $\mathcal{E}$, which according to Definition 3.9 is ESPSI($\mathcal{E}$) $= \|_{e\in E}\ P_e$ with

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

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

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

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

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

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

Theorem 4.37. *For an event structure $\mathcal{E}$ and an initial empty configuration $C = \emptyset$, we have:*

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

Proof. On the left side of the equality the process ESPSI($\mathcal{E}$) looks like $\|_{e\in E}\ \mathbf{case}\ \varphi_e : \bar{e}\langle e\rangle.(\!|\{e\}|\!)$, without any assertion process because the initial configuration is $C = \emptyset$. This process is embedded into dcrPsi, using Definition 4.36, as:

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

Since the assertion $\mathbf{1} = (\emptyset, \emptyset, \emptyset, 0)$ then $\mathbf{emb}^0(\mathbf{1}) \parallel \overline{m}\langle \mathbf{emb}^0(\mathbf{1})\rangle.\mathbf{0}$ becomes $(\!|\emptyset, \emptyset, E, 0|\!) \parallel \overline{m}\langle(\emptyset, \emptyset, E, 0)\rangle.\mathbf{0}$. Each of the P_e are translated as in Definition 4.36.

On the right side of the equality, the Definition 4.5 for **dcr** says that the DCR $\mathbf{dcr}(\mathcal{E}) = (E, M, <, \emptyset, \emptyset, \emptyset, \sharp \cup \{(e, e) \mid e \in E\})$ has the initial marking $M = (\emptyset, \emptyset, E)$. This means that the P_k, $k = 0$, generated by DCRPSI($\mathbf{dcr}(\mathcal{E}), 0$) is the same as what we had on the left side above, i.e., $(\!|\emptyset, \emptyset, E, 0|\!) \parallel \overline{m}\langle(\emptyset, \emptyset, E, 0)\rangle.\mathbf{0}$.

It remains to check that the processes P_e that Definition 4.15 of DCRPSI generates are the same as $\mathbf{emb}(P_e)$, when considering the empty sets that **dcr** generates. Recall that the P_e generated by DCRPSI, on the right of the equality, are:

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

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

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

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

where $e\rightarrow\!\%$ is the same as $\{e\} \cup \sharp e$. This shows it is equal with $\mathbf{emb}(P_e)$ since $\mathbf{emb}^e(\varphi_e)$ is, by Definition 4.33, the same as $(< e, \emptyset, e)$, and $\rightarrow\!\bullet = <$ in our case. □

We must make sure that the embedding from Definition 4.36 is correct in the sense that it preserves behaviour, as expressed below.

Proposition 4.38 *(Embedding preserves behaviour). For an event structure $\mathcal{E}$, then* ESPSI($\mathcal{E}$) *and* $\mathbf{emb}$(ESPSI($\mathcal{E}$)) *have the same behaviour, i.e.:*

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

Proof. In short, the proof follows from Lemma 4.34 that correlates the entailment relations, thus showing that whenever an event is enabled in the eventPsi-process then it is also enabled in its embedding.

Construct the bisimulation relation $\overset{\bullet}{\sim}$ between the states of the two event-labelled transition systems as follows; and then show that this respects the requirements of being a bisimulation. We take $\text{ESPSI}(\mathcal{E}) \overset{\bullet}{\sim} \mathbf{emb}(\text{ESPSI}(\mathcal{E}))$ to be the initial points. We continue the construction procedure exactly as in the proof of Theorem 4.26. Therefore, for any E' reachable from $\text{ESPSI}(\mathcal{E})$ by some event e, take the process P' reachable from $\mathbf{emb}(\text{ESPSI}(\mathcal{E}))$ by the same e, if one such exists, and put $E' \overset{\bullet}{\sim} P'$. The E' and P' are necessarily unique. In the proof all we are interested in is the structure of building this relation, which is given by the above procedure.

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

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

These are enough since the determinism and uniqueness of the construction of the relation $\overset{\bullet}{\sim}$ ensure that the reachable states E' and P' are bisimilar (for both statements), thus completing the requirements of Definition 4.25.

For the first statement we use Lemma 4.34.

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

5. Conclusions and outlook

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

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

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

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

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

5.1. Further work

Nevertheless, the encodings leave open issues. Firstly, a discrepancy remains between the interleaving semantics based on SOS rules of psi-calculi, and the non-interleaving nature of the two models we considered. Further investigations would look for a non-interleaving concurrency semantics for psi-calculi (with initial results presented as [51]). Secondly, it is not completely satisfactory that the behaviour of the dcrPsi-processes is observed by a somewhat intentionally constructed event-labelled transition semantics. An improvement could be to consider a more compositional definition of event structures and DCR graphs as considered in [46]. Thirdly, it would be interesting to look into adding responses to psi-calculi as introduced in DCR graphs, allowing to represent liveness/progress properties, continuing along the lines of the work in [52] for Transition Systems with Responses.

The next steps towards studying adaptable, distributed and mobile computational artefacts will be to consider cases of workflows identified in field studies within the CompArt project and the notion of run-time refinement and adaptation supported by DCR graphs as presented in [46]. By embedding DCR graphs in the richer framework of psi-calculi we anticipate being able to experiment with richer process models, e.g. representing locations, mobility and resources used by actors in workflows.

Remark 5.1 *(On infinite set of events).* If one would want to apply our encoding to infinite event structures then the set E may be infinite, hence the process and individual elements of **A** and **C** may be infinite terms (i.e., infinite sets). In the encoding produced by ESPSI, the conditions $\pi_L(\varphi_e)$ would be finite, because of the principle of finite causes of Definition 3.1 that

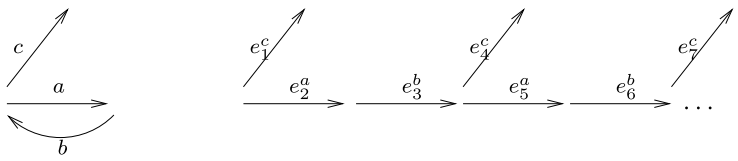

Fig. 8. Example of cancelling infinitely many different events coming from a loop unfolding. On the left we used a process graph view; on the right the event structures representation.

event structures respects. Still, the $\pi_R(\varphi_e)$ may be infinite, because there is no restriction on the conflict relation in event structures, and thus an event can be in conflict with infinitely many events.

A simple example where this would appear is pictured in Fig. 8, where we have a labelled transition system on the left and its unfolding as a labelled event structure on the right. The loop is unfolded into infinitely many sequential events, and for every second event we have a branch with a new c-labelled event which is in conflict with the rest of the infinite a, b labelled sequence. Executing one of these c-labelled events would mean cancelling all the infinitely many events that encode this branch. That is to say, the single event is in conflict with all the events on the looping branch, which are infinitely many.

Assertion terms from **A**, produced by ESPSI, are always finite because they encode, cf. Lemma 3.14, configurations, which are finite sets. Still, it is problematic to have the infinite right part of the conditions, since it is used in deciding the entailment relation where one needs to decide if the intersection of an infinite and finite set is empty.

Besides this, the encoding ESPSI would result in infinitely many parallel processes if the set of events is infinite, since a process is created for each $e \in E$. For practical use, infinite terms are not desirable, but for a theoretical encoding they could be fine, just like e.g. infinite summation in Milner's work on SCCS, infinite case construct for psi-calculi, or infinite conjunctions in some logics.

When building infinite nominal terms and infinite psi-processes one has to take care of not using infinitely many different free names, i.e., not to have infinite support for the nominal terms. Otherwise essential properties like alpha-renaming fail to work [53].

References

[1] M. Bravetti, C.D. Giusto, J.A. Pérez, G. Zavattaro, Adaptable processes, Log. Methods Comput. Sci. 8 (4) (2012), http://dx.doi.org/10.2168/LMCS-8(4:13)2012.
[2] M.D. Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, J. Mauro, Developing correct, distributed, adaptive software, in: Special Issue on New Ideas and Emerging Results in Understanding Software, Sci. Comput. Program. 97 (Part 1) (2015) 41–46, http://dx.doi.org/10.1016/j.scico.2013.11.019.
[3] J.C. Godskesen, T.T. Hildebrandt, Extending Howe's method to early bisimulations for typed mobile embedded resources with local names, in: R. Ramanujam, S. Sen (Eds.), Foundations of Software Technology and Theoretical Computer Science (FSTTCS), in: LNCS, vol. 3821, Springer, 2005, pp. 140–151.
[4] M. Bravetti, M. Carbone, T.T. Hildebrandt, I. Lanese, J. Mauro, J.A. Pérez, G. Zavattaro, Towards global and local types for adaptation, in: S. Counsell, M. Núñez (Eds.), Software Engineering and Formal Methods (SEFM), in: LNCS, vol. 8368, Springer, 2014, pp. 3–14.
[5] R.R. Mukkamala, T. Hildebrandt, T. Slaats, Towards trustworthy adaptive case management with dynamic condition response graphs, in: 17th IEEE International Enterprise Distributed Object Computing Conference (EDOC), IEEE, 2013, pp. 127–136.
[6] J. Bengtson, M. Johansson, J. Parrow, B. Victor, Psi-calculi: a framework for mobile processes with nominal data and logic, Log. Methods Comput. Sci. 7 (1) (2011), http://dx.doi.org/10.2168/LMCS-7(1:11)2011.
[7] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I–II, Inf. Comput. 100 (1) (1992) 1–77, http://dx.doi.org/10.1016/0890-5401(92)90008-4.
[8] A.M. Pitts, Nominal Sets: Names and Symmetry in Computer Science, Cambridge Tracts in Theoretical Computer Science, vol. 57, Cambridge Univ. Press, 2013.
[9] M. Abadi, A.D. Gordon, A calculus for cryptographic protocols: the spi calculus, Inf. Comput. 148 (1) (1999) 1–70, http://dx.doi.org/10.1006/inco.1998.2740.
[10] M. Abadi, C. Fournet, Mobile values, new names, and secure communication, in: C. Hankin, D. Schmidt (Eds.), 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, 2001, pp. 104–115.
[11] M.G. Buscemi, U. Montanari, CC-Pi: a constraint-based language for specifying service level agreements, in: R. De Nicola (Ed.), 16th European Symposium on Programming, Programming Languages and Systems (ESOP'07), in: LNCS, vol. 4421, Springer, 2007, pp. 18–32.
[12] R. De Nicola, G.L. Ferrari, U. Montanari, R. Pugliese, E. Tuosto, A process calculus for QoS-aware applications, in: J.-M. Jacquet, G.P. Picco (Eds.), 7th International Conference on Coordination Models and Languages (COORDINATION), in: LNCS, vol. 3454, Springer, 2005, pp. 33–48.
[13] M. Nielsen, G. Plotkin, G. Winskel, Petri nets, event structures and domains, in: Semantics of Concurrent Computation, in: LNCS, vol. 70, Springer, 1979, pp. 266–284.
[14] G. Winskel, Event structures, in: W. Brauer, W. Reisig, G. Rozenberg (Eds.), Advances in Petri Nets: Central Models and Their Properties, Part II, in: LNCS, vol. 255, Springer, 1987, pp. 325–392.
[15] T.T. Hildebrandt, R.R. Mukkamala, Declarative event-based workflow as distributed dynamic condition response graphs, in: K. Honda, A. Mycroft (Eds.), 3rd Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES), in: EPTCS, vol. 69, 2010, pp. 59–73.
[16] N. Wirth, Program development by stepwise refinement, Commun. ACM 14 (4) (1971) 221–227.
[17] R. van Glabbeek, U. Goltz, Refinement of actions and equivalence notions for concurrent systems, Acta Inform. 37 (4/5) (2001) 229–327, http://dx.doi.org/10.1007/s002360000041.
[18] T. Slaats, R. Mukkamala, T. Hildebrandt, M. Marquard, Exformatics declarative case management workflows as DCR graphs, in: F. Daniel, J. Wang, B. Weber (Eds.), Business Process Management, in: LNCS, vol. 8094, Springer, 2013, pp. 339–354.
[19] S. Debois, T. Hildebrandt, T. Slaats, M. Marquard, A case for declarative process modelling: agile development of a grant application system, in: G. Grossmann, S. Hallé, D. Karastoyanova, M. Reichert, S. Rinderle-Ma (Eds.), 18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOC Workshops), IEEE, 2014, pp. 126–133.

[20] H. Normann, C. Prisacariu, T. Hildebrandt, Concurrency models with causality and events as psi-calculi, in: I. Lanese, A. Lluch-Lafuente, A. Sokolova, H.T. Vieira (Eds.), 7th Interaction and Concurrency Experience (ICE 2014), in: Electronic Proceedings in Theoretical Computer Science (EPTCS), vol. 166, Open Publishing Association, 2014, pp. 4–20.
[21] H. Hüttel, Typed psi-calculi, in: J.-P. Katoen, B. König (Eds.), 22nd International Conference on Concurrency Theory (CONCUR'11), in: LNCS, vol. 6901, Springer, 2011, pp. 265–279.
[22] H. Hüttel, Types for resources in ψ-calculi, in: M. Abadi, A. Lluch-Lafuente (Eds.), 8th International Symposium on Trustworthy Global Computing (TGC), in: LNCS, vol. 8358, Springer, 2014, pp. 83–102.
[23] J. Riely, M. Hennessy, A typed language for distributed mobile processes (extended abstract), in: D.B. MacQueen, L. Cardelli (Eds.), Proceedings of the 25th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), ACM, 1998, pp. 378–390.
[24] M. Hennessy, A Distributed Pi-Calculus, Cambridge University Press, 2007.
[25] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
[26] R. Milner, Calculi for synchrony and asynchrony, Theor. Comput. Sci. 25 (1983) 267–310, http://dx.doi.org/10.1016/0304-3975(83)90114-7.
[27] M. Carbone, S. Maffeis, On the expressive power of polyadic synchronisation in pi-calculus, Nord. J. Comput. 10 (2) (2003) 70–98.
[28] J.A. Goguen, J. Meseguer, Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations, Theor. Comput. Sci. 105 (2) (1992) 217–273, http://dx.doi.org/10.1016/0304-3975(92)90302-V.
[29] C. Urban, A.M. Pitts, M. Gabbay, Nominal unification, Theor. Comput. Sci. 323 (1–3) (2004) 473–497, http://dx.doi.org/10.1016/j.tcs.2004.06.016.
[30] J. Borgström, R. Gutkovas, J. Parrow, B. Victor, J.Å. Pohjola, A sorted semantic framework for applied process calculi, Log. Methods Comput. Sci., http://arxiv.org/abs/1510.01044 (submitted).
[31] J. Borgström, R. Gutkovas, J. Parrow, B. Victor, J.Å. Pohjola, A sorted semantic framework for applied process calculi (extended abstract), in: M. Abadi, A. Lluch-Lafuente (Eds.), 8th International Symposium on Trustworthy Global Computing (TGC), in: Lecture Notes in Computer Science, vol. 8358, Springer, 2014, pp. 103–118.
[32] G. Winskel, Event structure semantics for CCS and related languages, in: M. Nielsen, E.M. Schmidt (Eds.), 9th Colloquium on Automata, Languages and Programming (ICALP), in: LNCS, vol. 140, Springer, 1982, pp. 561–576.
[33] V.R. Pratt, Modeling concurrency with geometry, in: POPL'91, 1991, pp. 311–322.
[34] V.R. Pratt, Higher dimensional automata revisited, Math. Struct. Comput. Sci. 10 (4) (2000) 525–548.
[35] R. van Glabbeek, On the expressiveness of higher dimensional automata, Theor. Comput. Sci. 356 (3) (2006) 265–290, http://dx.doi.org/10.1016/j.tcs.2006.02.012.
[36] R. van Glabbeek, G. Plotkin, Configuration structures, event structures and Petri nets, Theor. Comput. Sci. 410 (41) (2009) 4111–4159, http://dx.doi.org/10.1016/j.tcs.2009.06.014.
[37] C. Johansen, ST-structures, J. Log. Algebraic Methods Program. (2016), http://dx.doi.org/10.1016/j.jlamp.2015.10.009.
[38] V. Gupta, Chu spaces: a model of concurrency, Ph.D. thesis, Stanford University, 1994.
[39] V.R. Pratt, Chu spaces and their interpretation as concurrent objects, in: Computer Science Today: Recent Trends and Develop, in: LNCS, vol. 1000, Springer, 1995, pp. 392–405.
[40] G. Winskel, M. Nielsen, Models for concurrency, in: S. Abramski, D. Gabbay, T. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. 4, Oxford Univ. Press, 1995, pp. 1–148.
[41] R.J. van Glabbeek, F.W. Vaandrager, The difference between splitting in n and $n+1$, Inf. Comput. 136 (2) (1997) 109–142, http://dx.doi.org/10.1006/inco.1997.2634.
[42] P. Degano, R. De Nicola, U. Montanari, Partial orderings descriptions and observations of nondeterministic concurrent processes, in: J.W. de Bakker, W.P. de Roever, G. Rozenberg (Eds.), Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, REX Proceedings, in: LNCS, vol. 354, Springer, 1988, pp. 438–466.
[43] R. Milner, Functions as processes, Math. Struct. Comput. Sci. 2 (2) (1992) 119–141, http://dx.doi.org/10.1017/S0960129500001407.
[44] J. Aranda, C.D. Giusto, M. Nielsen, F.D. Valencia, CCS with replication in the Chomsky hierarchy: the expressive power of divergence, in: Z. Shao (Ed.), 5th Asian Symposium on Programming Languages and Systems (APLAS), in: LNCS, vol. 4807, Springer, 2007, pp. 383–398.
[45] T.T. Hildebrandt, R.R. Mukkamala, T. Slaats, Nested dynamic condition response graphs, in: F. Arbab, M. Sirjani (Eds.), 4th IPM International Conference on Fundamentals of Software Engineering (FSEN), in: LNCS, vol. 7141, Springer, 2012, pp. 343–350.
[46] S. Debois, T. Hildebrandt, T. Slaats, Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes, in: N. Bjørner, F. de Boer (Eds.), 20th International Symposium on Formal Methods (FM), in: LNCS, vol. 9109, Springer, 2015, pp. 143–161.
[47] T. Slaats, Flexible process notations for cross-organizational case management systems, Ph.D. thesis, IT University of Copenhagen, January 2015.
[48] G. Winskel, Events in computation, Ph.D. thesis, University of Edinburgh, 1980.
[49] M. Rao, T. Hildebrandt, J. Tøth, The resultmaker online consultant: from declarative workflow management in practice to LTL, in: 12th Enterprise Distributed Object Computing Conference Workshops, IEEE, 2008, pp. 135–142.
[50] J. Borgström, S. Huang, M. Johansson, P. Raabjerg, B. Victor, J.Å. Pohjola, J. Parrow, Broadcast psi-calculi with an application to wireless protocols, in: G. Barthe, A. Pardo, G. Schneider (Eds.), 9th International Conference on Software Engineering and Formal Methods (SEFM), in: LNCS, vol. 7041, Springer, 2011, pp. 74–89.
[51] H. Normann, C. Prisacariu, T. Hildebrandt, True concurrency semantics for psi-calculi, in: 1st International Workshop on Meta Models for Process Languages (MeMo), 2014 (presentation).
[52] M. Carbone, T.T. Hildebrandt, G. Perrone, A. Wasowski, Refinement for transition systems with responses, in: S.S. Bauer, J.-B. Raclet (Eds.), 4th Workshop on Foundations of Interface Technologies (FIT'12), in: EPTCS, vol. 87, 2012, pp. 48–55.
[53] M. Gabbay, A.M. Pitts, A new approach to abstract syntax with variable binding, Form. Asp. Comput. 13 (3–5) (2002) 341–363, http://dx.doi.org/10.1007/s001650200016.