https://doi.org/10.1007/s00165-021-00546-3
The Author(s) © 2021
Formal Aspects of Computing (2021) 33: 803–826

Formal Aspects of Computing

Denotational semantics of channel mobility in UTP-CSP

Gerard Ekembe Ngondi

Lero, the SFI Research Centre for Software,
Trinity College Dublin, Dublin, Ireland

Abstract. In this paper, we present the denotational semantics for channel mobility in the Unifying Theories of Programming (UTP) semantics framework. The basis for the model is the UTP theory of reactive processes, precisely, the UTP semantics for Communicating Sequential Processes (CSP), which is extended to allow the mobility of channels—the set of channels that a process can use for communication (its interface), originally static or constant (set during the process's definition), is now made dynamic or variable: it can change during the process's execution. A channel is thus moved around by communicating it via other channels and then allowing the receiving process to extend its interface with the received channel. We introduce a new concept, the *capability* of a process, which allows separating the ownership of channels from the knowledge of their existence. Mobile processes are then defined as having a static capability and a dynamic interface. Operations of a mobile telecommunications network, e.g., handover, load balancing, are used to illustrate the semantics. We redefine CSP operators and in particular provide the first semantics for the renaming and hiding operators in the context of channel mobility.

Keywords: Channel mobility, Dynamic network (topology), Denotational semantics, Unifying theories of programming, UTP, Communicating sequential processes, CSP

1. Introduction

In a network, channel mobility is concerned with the movement of a channel from one component to another, where channels are used by components to exchange messages. We will use the term dynamic network system (or simply dynamic network) to refer to mobile systems in which channels can move. By symmetry, we will use the term static system to refer to a static network system (or simply static network).

Pi-calculus [Mil99] is the first process algebra for reasoning about channel mobility. Pi-calculus represents programs as processes, whose semantics are operational; channel mobility is achieved by sending channels as messages through other channels, from one process to another. CSP [Hoa85, Ros10] is another popular process algebra for reasoning about concurrency in general, and networks in particular. However, CSP is limited to static networks. CSP has both operational and denotational semantics.

Correspondence to: Gerard Ekembe Ngondi, E-mail: gerard.ekembe@tcd.ie

We are interested in CSP denotational semantics, precisely, UTP-CSP [HoaHe98, Chap. 8], an alternative presentation of CSP based on Unifying Theories of Programming (UTP) [HoaHe98], a framework whose aim is to unify the semantics of various programming languages and paradigms. UTP notably extends CSP with a richer notion of state and provides a mechanism for linking UTP theories using notions of Galois connections [HoaHe98, Chap. 5].

In this paper, we present mobile CSP, which extends static UTP-CSP [HoaHe98, CavWoo06] with the semantics of channel mobility. In mobile CSP, channels are treated as *concrete* entities and can thus be moved as messages between processes. Traces are enriched to record, in addition to the events history, the interface history of a process. We call the new traces dynamic alphabetised traces (DATs). We introduce a new concept, the *capability* of a process, which represents the universal set of channels and models the statement *a process knows of the existence of channels (outside its interface)*. This allows us to refine the meaning of the *interface* as modelling the statement *a process owns a given channel* (in its interface). Mobile processes are characterised as having a static capability and a dynamic interface, formalised by new healthiness conditions. We redefine static CSP operators to reflect the changes in semantics due to channel mobility. The main updates concern input and output prefix operations, and parallel composition. Other processes such as assignment are updated only with regard to healthiness conditions. Two operators are particularly difficult to define in the presence of channel mobility, namely renaming and hiding. We present the first definition of static renaming and static hiding in the context of channel mobility—"static" here indicates that the operators work as in static CSP, i.e., the operators apply to the interface of a process provided initially, before any execution of the process.

This paper significantly extends [Ek16a], which contains many elements of the formalisation reproduced here. The following elements are presented in this paper and not in [Ek16a]: many operators, notably parallel composition, renaming and hiding; all the examples; a new healthiness condition, ***MC4***.[1]

The next section briefly introduces the semantics of UTP-CSP processes. Section 3.1, presents the formalisation of dynamic network systems in UTP; Sect. 3.2 contains the semantics of mobile processes, Sect. 3.2.4 presents static renaming, and Sect. 3.2.5 static hiding. We discuss refinement and issues related to the interrupt operator in Sect. 4. Finally, in Sect. 5, we compare our results and their implications to related works, followed by the Conclusion, in Sect. 6.

2. Background: unifying theories of programming (UTP)

2.1. Generalities

UTP is a formal semantics framework for reasoning about programs, programming theories and the links or encodings between theories. The semantics of a program are given by a relation between the initial (undecorated) and final (decorated) observations that can be made of the variables that characterise the program behaviour.

A UTP theory is a collection of predicates and consists of three elements: an *alphabet*, containing only those variables that the predicates of the theory may mention; a *signature*, which contains the operators of the theory, and *healthiness conditions*, which are laws constricting the set of legal predicates to those that obey the properties expressed by the conditions. Healthiness conditions determine hence what predicate belongs to (can be defined/implemented in) a theory.

Healthiness conditions generally have the form: ***NAME*** $\quad P = f(P)$, for some idempotent function f (i.e., $f \circ f(x) = f(x)$). ***NAME*** stands for the name of the healthiness condition and is also used as an alias for f, i.e., we write $P =$ ***NAME***(P) and we say that P is ***NAME***-healthy.

A number of programming constructs and paradigms have been formalised using UTP. The most basic UTP theory is the theory of *Relations* [HoaHe98, Chap. 2], which notably allows us to specify most of the constructs of sequential programming (cf. Appendix A).

In what follows we present the UTP theory of *Reactive Processes*, which allows representing concurrent programs and serves as a basis for defining UTP-CSP.

[1]The condition defined by ***MC4*** is mentioned but left implicit in [Ek16a].

2.2. Reactive processes

The UTP theory of Reactive Processes [HoaHe98, CavWoo06] permits modelling programs that may interact with their environment. Reactive programs are defined as processes, i.e., predicates that allow us to characterise the intermediate states of a program, between initialisation and termination.

The alphabet of a reactive process consists of the following:

- $\mathcal{A}$, the set of authorised events or actions set ; $tr, tr' : \mathcal{A}^*$, the trace ; $ref, ref' : \mathbb{P}\,\mathcal{A}$, the refusal set
- $ok, ok' : \mathbb{B}$, stability and termination ; $wait, wait' : \mathbb{B}$, waiting states
- v, v', other variables

Reactive processes must also satisfy the following healthiness conditions:

$$\begin{array}{lll} \boldsymbol{R1} & P = P \wedge tr \le tr' & \\ \boldsymbol{R2} & P = \bigsqcap_s \{P[s, s \frown (tr' - tr)/tr, tr'] \mid s \in \mathcal{A}^*\} & \\ \boldsymbol{R3} & P = (II_{CSP} \lhd wait \rhd P) & \textbf{where} \end{array}$$

$$II_R \mathrel{\widehat=} (ok' = ok \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge v' = v) \lhd ok \rhd (tr \le tr')$$

R1 states that the occurrence of an event cannot be undone, viz., the trace can only get longer. ***R2*** states that the initial value of tr may not affect the current observation. ***R3*** states that a process does nothing when its predecessor has not yet terminated. II_{CSP} is the process that changes nothing: If not started, $ok = false$, then only trace expansion can be observed; otherwise the values of the variables remain unchanged.

A reactive process is one that satisfies all three healthiness conditions above. We also say that it satisfies the healthiness condition $\boldsymbol{R} = \boldsymbol{R1} \circ \boldsymbol{R2} \circ \boldsymbol{R3}$. Note that the order of the composition is irrelevant (cf. [CavWoo06]). A particular model for reactive processes is provided by the CSP process algebra [Hoa85, Ros98], presented subsequently.

2.3. UTP-CSP processes: syntax and semantics

UTP-CSP processes (or simply CSP processes below) are reactive processes that obey the following additional healthiness conditions:

$$\begin{array}{lll} \boldsymbol{CSP1} & P = P \lhd ok \rhd tr \le tr' & \\ \boldsymbol{CSP2} & P = P \mathbin{;} J & \textbf{where} \end{array}$$

$$J \mathrel{\widehat=} (ok \Rightarrow ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge v' = v)$$

CSP1 states that if a process has not started ($ok = false$) then nothing except for trace expansion can be said about its behaviour. Otherwise the behaviour of the process is determined by its definition. ***CSP2*** states that a process can always terminate. It characterises the fact that divergence can never be required.

A CSP process is one that satisfies the healthiness conditions ***R***, ***CSP1*** and ***CSP2***. We also say that it satisfies the healthiness condition $\boldsymbol{CSP} = \boldsymbol{R} \circ \boldsymbol{CSP1} \circ \boldsymbol{CSP2}$. Again, the order of the composition is irrelevant. The syntax and semantics of some CSP processes are given in Appendix B.

CSP processes permit the representation of static systems only, i.e., systems whose network topology does not change during their activation. For that reason the model for CSP processes presented so far will be referred to as the *static model* of CSP or static CSP. In order to represent mobile systems, a number of changes need to be provided to that static model; they are presented hereafter.

3. Mobile UTP-CSP

In what follows, we present mobile CSP. The mobility model has three main characteristics:

- channels are 'localised' in processes (viz., their interface);
- channels can be communicated as messages amongst processes, hence channels are concrete entities;
- the interface of a process (denoted by $\mathcal{I}$ hereafter) changes as a consequence of channel mobility.

First, we discuss the formalisation of dynamic networks in UTP.

3.1. Dynamic (network) systems - concepts and their formalisation

3.1.1. Definition of concepts

We will study channel mobility from the point of view of the processes involved, not that of the specifier. We assume that channels are unique, thus, a name clash between a process and its environment should not occur. If it occurs, then this will be considered an error. The Internet Protocol addressing schemes IPv4 and IPv6 and object identification in Object-Oriented Programming illustrate systems in which system elements, IP addresses and object identifiers respectively, are unique.

Capability vs. Interface. In UTP-CSP, the interface of a process is a constant. In particular, it models the statement *a process owns a channel*, viz., the process can actually use that channel for its communications. Channel mobility implies that the interface of mobile systems must be a variable: it changes with the movement of channels. Let us talk of either static interface or dynamic interface, accordingly.

Where do *new* channels come from? For static systems, all the channels that are owned are defined for once: in the interface. Since new channels are not owned yet, we introduce a new set called the *capability* of a process to define channels that 'might' be moved. We thus separate *ownership*, entirely determined by the interface, from *knowledge-of existence*, determined by the capability. Unlike the interface, the capability is static.

Definition 3.1.1 **(Capability** - MCh) *Let MCh denote the* capability *of a process, the set of channels whose existence is known by the process. Its value cannot change.* □

In this paper, we will assume that every mobile CSP process has the same capability. This implies that there is no limit to what channel a process can acquire. This assumption may be broken without loss of generality, viz., it would not fundamentally affect the semantics presented subsequently.

Channel names. In static CSP, channel names are just *abstract identifiers* in the sense that a channel name $ch \in \mathcal{I}$ represents/models a logical concept, 'the occurrence of a communication on the channel named *ch*', and not the channel itself. For channel mobility, channels must rather be modelled explicitly, as data elements: they will also be represented by channel names. For this purpose, it suffices to override the interface of processes to now contain concrete channels. The following variable will denote the interface of a mobile process.

Definition 3.1.2 *(mChans) $mChans, mChans' : \mathbb{P}\, MCh$, is the variable that contains the set of channels that have been acquired before the current observation, and are hence authorised. $mChans'$ contains the channels that will be authorised next.*

$mChans$ (resp. $mChans'$) denotes the dynamic interface *of a process.* □

Events of mobile channels. In static CSP, the interface of a process can be obtained only from its actions set $\mathcal{A}$, i.e., $\mathcal{I} = \{ch \mid \exists\, e \bullet ch.e \in \mathcal{A}\}$. With channel mobility, on the contrary, we start with the channels since they are the ones that may be moved, and then we obtain the corresponding set of events. Hence, we define the set *MCev* that contains events related with mobile channels only, i.e., events of the form $c.m$ where $c \in MCh$.

Definition 3.1.3 *(MCev) Let MCev denote the set of events (or actions set) obtained from MCh.*

$$MCev \;\widehat{=}\; \{ch.e \mid ch \in MCh\}$$ □

In static CSP, the empty trace $\langle\rangle$ records that no event has occured. In mobile CSP, it will be more convenient to represent the null event explicitly.

Definition 3.1.4 (Null event) *Let nil denote the null event. It represents that no event has occured.* □

That is, both $\langle\rangle$ and $\langle nil \rangle$ denote the empty trace (were *nil* used in static CSP). The following definition summarises the actions set for mobile processes, denoted by Σ.

Definition 3.1.5 *(Σ) Let $\mathcal{A}$ denote the actions of a process, viz., it does include communication events from static channels only and other kinds of events. Let Σ denote the actions set for mobile processes, then:*

$$\Sigma \mathrel{\widehat{=}} \{nil\} \cup \mathcal{A} \cup MCev \qquad \textbf{\textit{where}} \quad \mathcal{A} \cap MCev = \{\}$$ □

The previous definition assumes the possibility of defining some channels as static, but this is not essential and is not enforced in the subsequent semantics.
Dynamic alphabetised traces. Let *dtr* denote the trace associated with acquired mobile channels, i.e., those in *mChans*. The value of *mChans* at a given time defines which events may be recorded at that time; at different times, *mChans* may have different values: *dtr* must reflect such changes. We thus introduce the notion of *dynamic alphabetised trace.*

Definition 3.1.6 (DAT) *A* dynamic alphabetised trace *or* DAT *is any trace of the form $\langle ..., (s, e), ... \rangle$ where s is the valid dynamic interface (viz., given by mChans) at the time of the observation, and e is the event recorded at that time.* □

Definition 3.1.7 (DAT of a mobile process) dtr, $dtr' : (\mathbb{P}(MCh) \times \Sigma)^*$, *denotes the dynamic alphabetised trace of a mobile process.* □

Notation. Subsequently, the following two projections are used to select each component of an element in a DAT trace: $\pi_1(s, e) = s$, $\pi_2(s, e) = e$. They are overridden to get also the first and second component of all elements in a trace, respectively. Let $k \in \{1, 2\}$, then

$$\pi_k(\langle (s, e) \rangle) \mathrel{\widehat{=}} \langle \pi_k(s, e) \rangle$$
$$\pi_k(head\ dtr \frown tail\ dtr) \mathrel{\widehat{=}} \pi_k(head\ dtr) \frown \pi_k(tail\ dtr)$$

Initial interface. The initial value of the interface of a process correpsonds to the first value of the interface that is recorded in any trace of the process.

Definition 3.1.8 (Initial interface, $mChans^k$) *Let $mChans^k$ ($k \in \mathbb{N}$) denote the k-th interface element of a given DAT dtr′. Then:*

$$mChans^k = \pi_1(dtr'[k])$$ □

$$mChans^0 = \pi_1(first\ dtr') = \pi_1(dtr'[0]) \qquad \textit{is the initial interface of a process.}$$

Refusals. Let *dref* denote the refusals set for mobile processes. The static type for *dref* will be $\mathbb{P}\,\Sigma$. Then, it is necessary to restrict the value of *dref* such that owned events only may be refused. This can be achieved by means of a healthiness condition (cf. ***MC3*** below).

Definition 3.1.9 (Refusals) $dref, dref' : \mathbb{P}\,\Sigma$, *denotes the refusals set of a mobile process.* □

Snapshots. The observation of a dynamic network may be divided according to its different topologies. A process is said to have a static (or fixed) network/topology when its interface is the same whatever the elements of its DAT. Formally:[2]

Definition 3.1.10 ***(SN)***

$$\textbf{\textit{SN}} \quad P = P \wedge \forall (s_1, e_1), (s_2, e_2) : \mathbb{P}\,MCh \times \Sigma \mid \#\ dtr' \geq 2 \bullet\ (s_1, e_1) \frown (s_2, e_2) \in dtr' \Rightarrow s_1 = s_2$$

P has a static network topology or simply P is static if $P = \textbf{SN}(P)$. That is, $\forall\, k : mChans^k = mChans^{k+1}$.
□

[2]$\#\ s$ denotes the number of elements of the sequence s.

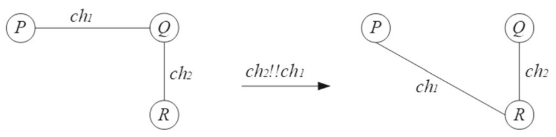

Fig. 1. Channel Mobility with 3 processes. (left) Before the migration of ch_1. (right) After the migration of ch_1

A process must have at least two distinct snapshots (viz., must be the concatenation of at least two distinct ***SN*** processes) to be considered of having a dynamic topology. In other words, at least two consecutive elements of its trace must have different (but not disjoint) interfaces. Formally:

Definition 3.1.11 ***(DN)***

$$\boldsymbol{DN} \quad P = P \wedge \exists\,(s_1, e_1), (s_2, e_2) : \mathbb{P}\, MCh \times \Sigma \bullet \ (s_1, e_1) ^\frown (s_2, e_2) \in dtr' \Rightarrow s_1 \subset s_2 \vee s_2 \subset s_1$$

P has a dynamic network topology or simply P is dynamic if $P = \boldsymbol{DN}(P)$. That is, $\exists\, k : mChans^k \neq mChans^{k+1}$, where $mChans^k$ and $mChans^{k+1}$ are not disjoint. □

Example 3.1.1 *Consider the network of three processes P, Q and R connected as shown in Fig. 1(left). Another possible topology for such a network can be obtained by removing the link ch_1 between P and Q, then using ch_1 to connect P and R instead, as shown in Fig. 1(right). The left and right networks define two snapshots.* □

3.1.2. Healthiness conditions

The guarantee that *a process must use only channels that it already owns* is expressed by the following healthiness condition:

Definition 3.1.12 ***(MC1)***

$$\boldsymbol{MC1} \quad P \ = \ P \wedge \forall\, s : \mathbb{P}\, MCh, e : \Sigma \bullet (s, e) \in dtr' \Rightarrow e \in s$$ □

MC1 states that every event e that is recorded must belong to the dynamic interface s (the associated actions set) valid at the time of the recording.

By definition, the mobility of a single channel (or of many together) induces a snapshot dichotomy between the topology before the movement and the one after. So, for a trace $\langle (s_1, e_1), (s_2, e_2)\rangle$, where $s_1 \neq s_2$, $\langle (s_1, e_1)\rangle$ would belong to the first snapshot and $\langle (s_2, e_2)\rangle$ to the second. That is, in a single step, it should only be possible to release the whole of the actual interface at once, but not to acquire a new channel at the same time. This is expressed by the following healthiness condition:

Definition 3.1.13 ***(MC2)***

$$\boldsymbol{MC2} \quad P = P \ \wedge\ \forall (s_1, e_1), (s_2, e_2) : \mathbb{P}\, MCh \times \Sigma \mid \#\ dtr' \geq 2 \bullet (s_1, e_1) ^\frown (s_2, e_2) \in dtr' \Rightarrow (s_1 \subseteq s_2 \ \vee\ s_2 \subseteq s_1)$$ □

Example 3.1.2 *The interface histories $\langle \{ch_1, ch_2\}, \{ch_2, ch_3, ch_4\}\rangle$ and $\langle \{ch_1, ch_2\}, \{ch_3, ch_4\}\rangle$ are forbidden by **MC2**; however, $\langle \{ch_1, ch_2\}, \{ch_2\}, \{ch_2, ch_3, ch_4\}\rangle$ is a valid interface history.* □

MC2 also translates the idea that the dynamic interface is always fixed (or completely determined) before entering a new snapshot, and that it is the previous snapshot (process) that fixes it.

The healthiness condition expressing that owned events only can be refused is given below.

Definition 3.1.14 *(**MC3**)* $\textbf{\textit{MC3}} \quad P \; = \; P \wedge \mathit{dref}' \subseteq (\mathcal{A} \cup \mathit{mChans}')$ $\square$

Note that mChans' is used above for economy of notation, to denote the corresponding set of events, say mCev'. ***MC3*** corresponds to a law on refusals for static CSP processes (cf. [Hoa85, §3.4, L8]).[3]

Earlier we introduced the *nil* notation to denote that no event has occured. The events history of a process can be empty. In contrast, the interface history *should not* be empty as we expect it to contain at least one element, the value of the initial interface. What about the final value of the interface history? Initially, $\mathit{mChans} = \mathit{mChans}' = \mathit{last}\,\pi_1(\mathit{dtr}')$, i.e., the last value of the interface history, $\mathit{last}\,\pi_1(\mathit{dtr}')$, coincides with mChans'. However, it is possible to acquire a new channel, say *newch*, without simultaneously updating the interface history, as in $\langle ..., (\{in\}, in.newch)\rangle$; whereas a simultaneous update would give the trace $\langle ..., (\{in\}, in.newch), (\{in, newch\}, nil)\rangle$ instead. The condition $\mathit{mChans}' = \mathit{last}\,\pi_1(\mathit{dtr}')$ implies that the interface history must mirror the value of *mChans* at all times, which is desirable. The values of mChans' and $\pi_1(\mathit{dtr}')$ would thus be redundant. If we try and remove the redundancy by eliminating $\mathit{mChans}, \mathit{mChans}'$ from our model, the previous condition would need to be changed. We think it better to keep the history separate from what it records, so we will keep using $\mathit{mChans}, \mathit{mChans}'$.

Definition 3.1.15 *(**MC4**)* $\textbf{\textit{MC4}} \quad P \; = \; P \wedge \mathit{mChans}' = \mathit{last}\,\pi_1(\mathit{dtr}')$ $\square$

DATs lead us to reconsider the healthiness conditions ***R1***, ***R2***, ***R3***, ***CSP1***, and ***CSP2*** (cf. § 2.2). The main issue here is substituting the symbols $tr, tr', \mathit{ref}, \mathit{ref}'$ for $\mathit{dtr}, \mathit{dtr}', \mathit{dref}, \mathit{dref}'$, respectively. Semantically however, there is not much change. For readability, each ***Rk*** healthiness condition will be renamed ***RkM***.

Definition 3.1.16 *(**CSPM**)*

$$
\begin{array}{rrcl}
\textbf{\textit{R1M}} & P & = & P \wedge \mathit{dtr} \le \mathit{dtr}' \qquad \square \\
\textbf{\textit{R2M}} & P & = & \bigsqcap \left\{ P[t, \left(t \frown (\mathit{dtr}' - \mathit{dtr})\right) / \mathit{dtr}, \mathit{dtr}'] \mid t \in (\mathbb{P}\, \mathit{MCh} \times \Sigma)^* \wedge \pi_1(t) = \pi_1(\mathit{dtr}) \right\} \\
\textbf{\textit{R3M}} & P & = & (\mathit{II}_{\mathit{Rm}} \lhd \mathit{wait} \rhd P) \qquad \textbf{\textit{where}} \\
 & \mathit{II}_{\mathit{Rm}} & \widehat{=} & (ok' = ok \wedge \mathit{wait}' = \mathit{wait} \wedge \mathit{dtr}' = \mathit{dtr} \wedge \mathit{dref}' = \mathit{dref} \wedge v' = v) \lhd ok \rhd (\mathit{dtr} \le \mathit{dtr}') \\
\textbf{\textit{CSP1M}} & P & = & P \lhd ok \rhd \mathit{dtr} \le \mathit{dtr}' \\
\textbf{\textit{CSP2M}} & P & = & P \,;\, J \qquad \textbf{\textit{where}} \quad J \;\widehat{=}\; (ok \Rightarrow ok' \wedge \mathit{wait}' = \mathit{wait} \wedge \mathit{dtr}' = \mathit{dtr} \wedge \mathit{dref}' = \mathit{dref} \wedge v' = v)
\end{array}
$$

Definition 3.1.17 *(**MCr1**, **MCr3**, **MCcsp1**) Let $mtr = \pi_1(dtr)$ denote the interface history of a process, let $tr = \pi_2(dtr)$ denote its events history. Then, when replacing dtr by tr in the equations above (Def. 3.1.16), we obtain the traditional CSP healthiness conditions. We also obtain three new healthiness conditions when replacing dtr by mtr in the condition $dtr \le dtr'$, namely:*

$$
\begin{array}{rrcl}
\textbf{\textit{MCr1}} & P & = & P \wedge \mathit{mtr} \le \mathit{mtr}' \qquad \square \\
\textbf{\textit{MCr3}} & P & = & (\mathit{II}_{\mathit{mtr}} \lhd \mathit{wait} \rhd P) \qquad \textbf{\textit{where}} \\
 & \mathit{II}_{\mathit{mtr}} & \widehat{=} & (ok' = ok \wedge \mathit{wait}' = \mathit{wait} \wedge \mathit{mtr}' = \mathit{mtr} \wedge \mathit{dref}' = \mathit{dref} \wedge v' = v) \lhd ok \rhd (\mathit{mtr} \le \mathit{mtr}') \\
\textbf{\textit{MCcsp1}} & P & = & P \lhd ok \rhd \mathit{mtr} \le \mathit{mtr}'
\end{array}
$$

MCr1 means that the interface history only ever grows. ***MCr3*** means that when a process is waiting for its predecessor to terminate ($\mathit{wait} = \mathit{true}$) and the predecessor is not stable ($ok = \mathit{false}$), then we can observe the interface history growing. ***MCcsp1*** means that when a process is not stable ($ok = \mathit{false}$), we can yet observe its interface history growing.

[3] $X \in \mathit{refusals}(P) \Rightarrow X \in \mathcal{A}P$ where X is a non-empty set of events, P a process, $\mathit{refusals}(P)$ the set of refusals of P, $\mathcal{A}P$ the actions set of P.

Consequence 1 **(Relation** $\boldsymbol{RkM} - \boldsymbol{Rk}$**)**

$$\begin{aligned} \boldsymbol{R1M} &= \boldsymbol{R1} \circ \boldsymbol{MCr1} \\ \boldsymbol{R3M} &= \boldsymbol{R3} \circ \boldsymbol{MCr3} \\ \boldsymbol{CSP1M} &= \boldsymbol{CSP1} \circ \boldsymbol{MCcsp1} \end{aligned}$$

Proof $dtr \le dtr' \Leftrightarrow mtr \le mtr' \wedge tr \le tr'$. □

Theorem 3.1.1 *Let* $\boldsymbol{CSPM} = \boldsymbol{R1M} \circ \boldsymbol{R2M} \circ \boldsymbol{R3M} \circ \boldsymbol{CSP1M} \circ \boldsymbol{CSP2M}$. *All* $\boldsymbol{CSPM}$ *processes are* $\boldsymbol{CSP}$ *processes, viz.*

$$\boldsymbol{CSPM} = \boldsymbol{CSP} \circ \boldsymbol{MCr1} \circ \boldsymbol{MCr3} \circ \boldsymbol{MCcsp1}$$

Proof From Consequence 1. □

Corollary 1 *All mobile processes are* $\boldsymbol{CSP}$*-healthy.*

Proof By Def. 3.2.1, all mobile processes are $\boldsymbol{CSPM}$, and all $\boldsymbol{CSPM}$ processes are $\boldsymbol{CSP}$, by Thrm. 3.1.1. Hence all mobile processes are $\boldsymbol{CSP}$. □

We now provide some important properties of the previous healthiness conditions.

Theorem 3.1.2 *(1)* $\boldsymbol{MC1}$, $\boldsymbol{MC2}$, $\boldsymbol{MC3}$, $\boldsymbol{MC4}$, *are all conjunctive, hence they are all monotonic, idempotent, pairwise commutative, and* $\{\wedge, \vee, \lhd b \rhd, \fatsemi, \mu\}$*-closed. (2) Furthermore, they are all pairwise commutative with* $\boldsymbol{R1M}$, $\boldsymbol{R2M}$, $\boldsymbol{R3M}$, $\boldsymbol{CSP1M}$ *and* $\boldsymbol{CSP2M}$*.*

Proof (1) is a consequence of [Harw08, Theorems 1,2,3]—see also Appendix C. In (2), commutativity with $\boldsymbol{R1M}$, $\boldsymbol{R2M}$, and $\boldsymbol{CSP2M}$ comes from the latter being all conjunctive. Commutativity with $\boldsymbol{R3M}$ and $\boldsymbol{CSP1M}$ comes from the closure and idempotence of conjunctive healthiness conditions. □

3.2. The semantics

In this section we present the denotational semantics of channel mobility. As every UTP theory, it must have three elements: an alphabet, a signature and healthiness conditions. The highlight of this section is the semantics of the operations that may change the interface of a process during its activation, channel-passing input and output prefixes. We also provide the first semantics for renaming and hiding in the context of channel mobility.

The following definition summarizes the previous discussions.

Definition 3.2.1 **(Mobile processes)** *A mobile process is one that satisfies the healthiness conditions* $\boldsymbol{R1M}$, $\boldsymbol{R2M}$, $\boldsymbol{R3M}$, $\boldsymbol{CSP1M}$, $\boldsymbol{CSP2M}$, $\boldsymbol{MC1}$, $\boldsymbol{MC2}$, $\boldsymbol{MC3}$, *and* $\boldsymbol{MC4}$, *and has an alphabet defined by* $\{v, v'\} \cup \{\mathbf{o}, \mathbf{o}' \mid \mathbf{o} \in \{ok, wait, dtr, dref, mChans\}\} \cup MCh \cup \mathcal{A} \cup MCev \cup \{nil\}$. □

3.2.1. Some mobile processes

Assignment, SKIP. The following definitions are similar to their static CSP formulation, but made healthy for mobile CSP. In particular, local assignments to $mChans$ are forbidden as the interface should grow and shrink as the result of channel passing communications only (see below). Hence $mChans' = mChans$ and $\boldsymbol{MC4}$ holds. $\boldsymbol{MC3}$ ensures that we get the right value of $dref, dref'$. Since $dtr' = dtr$, no modification of the value of the trace is possible, hence assignment is vacuously $\boldsymbol{MC1}$ and $\boldsymbol{MC2}$ healthy.

Definition 3.2.2

$$(x := e) \;\widehat{=}\; \boldsymbol{MC3} \circ \boldsymbol{R3M} \circ \boldsymbol{CSP1M}(ok' \wedge \neg\, wait' \wedge dtr' = dtr \wedge mChans' = mChans \wedge x' = e \wedge v' = v)$$
□

$$SKIP \;\widehat{=}\; \exists\, dref \bullet II_{Rm}$$

Theorem 3.2.1 *Def. 3.2.2 obeys all assignment laws. SKIP obeys all skip laws [Hoa85]–see also Appendices A and B.*

Proof One way of proving this theorem may be to apply Def. 3.2.2 to every law in Appendices A and B and see if it holds. Here instead, we show that our definition is *additive* in the sense of Abrial [Abr84], then it preserves all the laws of the formula that it extends.

Let $[\![\, P \,]\!]_{th}$ denote the encoding or semantics of a given predicate P in a given UTP theory *th*. Let *rel* and *csp* denote the UTP theories of Relations [HoaHe98, Chap. 2] and CSP [HoaHe98, Chap. 8]–see also Sect. 2; and let *mob* denote mobile CSP. We have:

$$\begin{aligned}
[\![\, x := e \,]\!]_{csp} &= \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge x' = e \wedge v' = v)\\
&= \boldsymbol{R3} \circ \boldsymbol{CSP1}(x' = e \wedge v' = v) \wedge \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr)\\
&= \{\boldsymbol{R3}\,\text{and}\,\boldsymbol{CSP1}\,\text{are not conditions on}\,x, x', v, v'\}\\
&= (x' = e \wedge v' = v) \wedge \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr)\\
&= [\![\, x := e \,]\!]_{rel} \wedge \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr)
\end{aligned}$$

That is, for assignment, *csp* extends *rel*, viz., there is an *additive* transformation that extends theory *rel* to yield theory *csp*. We now see how this implies law-preservation.

$[\![\, x := e \,]\!]_{rel} = [\![\, x, y := e, y \,]\!]_{rel}$	$[\![\, x := e \,]\!]_{rel} \mathbin{;} [\![\, x := f(x) \,]\!]_{rel} = [\![\, x := f(e) \,]\!]_{rel}$
$\Leftrightarrow$ {propositional calculus}	$\Leftrightarrow$ {seq. comp. def. , propositional calculus}
$[\![\, x := e \,]\!]_{rel} \wedge \phi = [\![\, x, y := e, y \,]\!]_{rel} \wedge \phi$	$([\![\, x := e \,]\!]_{rel} \wedge \phi) \mathbin{;} [\![\, x := f(x) \,]\!]_{rel} = [\![\, x := f(e) \,]\!]_{rel} \wedge \phi$

Let $\phi = \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr)$, then $[\![\, x := e \,]\!]_{csp} = [\![\, x := e \,]\!]_{rel} \wedge \phi$. In words, the laws about $[\![\, x := e \,]\!]_{csp}$ can be derived from those of $[\![\, x := e \,]\!]_{rel}$. There is thus no need to prove the laws of $[\![\, x := e \,]\!]_{rel}$ for $[\![\, x := e \,]\!]_{csp}$. This is the essence of additive definitions [Abr84].

We show similarly that, for assignment, *mob* extends *csp*. Recall, $mtr = \pi_1(dtr)$, $tr = \pi_2(dtr)$.

$$\begin{aligned}
&[\![\, x := e \,]\!]_{mob}\\
&= \{\text{CSPmob.assign.def}\,3.2.2\}\\
&\boldsymbol{MC3} \circ \boldsymbol{R3M} \circ \boldsymbol{CSP1M}(ok' \wedge \neg\, wait' \wedge dtr' = dtr \wedge mChans' = mChans \wedge x' = e \wedge v' = v)\\
&= \{\text{conj.health.cond., propositional calculus}\}\\
&\boldsymbol{MC3} \circ \boldsymbol{R3M} \circ \boldsymbol{CSP1M}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge x' = e \wedge v' = v) \wedge\\
&\qquad \boldsymbol{MC3} \circ \boldsymbol{R3M} \circ \boldsymbol{CSP1M}(mtr' = mtr \wedge mChans' = mChans)\\
&= \{\boldsymbol{R3M} = \boldsymbol{R3} \circ \boldsymbol{MCr3}, \boldsymbol{CSP1M} = \boldsymbol{CSP1} \circ \boldsymbol{MCcsp1}, \boldsymbol{MC3}\,\text{conjunctive}\}\\
&\boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge x' = e \wedge v' = v) \wedge\\
&\qquad \boldsymbol{MC3} \circ \boldsymbol{MCr3} \circ \boldsymbol{MCcsp1}(mtr' = mtr \wedge mChans' = mChans)\\
&= \{\text{CSP.assign.def}\}\\
&[\![\, x := e \,]\!]_{csp} \wedge \boldsymbol{MC3} \circ \boldsymbol{MCr3} \circ \boldsymbol{MCcsp1}(mtr' = mtr \wedge mChans' = mChans)
\end{aligned}$$

From what precedes, we trivially have:

$$[\![\, SKIP \,]\!]_{mob} = [\![\, SKIP \,]\!]_{csp} \wedge \boldsymbol{MC3} \circ \boldsymbol{MCr3} \circ \boldsymbol{MCcsp1}(mtr' = mtr \wedge mChans' = mChans)$$

□

Unchanged definitions. The semantics of *STOP* and *CHAOS* remain unchanged. The semantics of the following operators are also unchanged: iteration ($b * P$), substitution ($P[e/x]$), sequential composition ($;$), conditional ($\lhd b \rhd$), internal choice ($\sqcap$), and external choice ($\Box$)—cf. Appendix B.

Prefix. In static CSP, the occurrence of an action a is denoted by the predicate $do_{\mathcal{A}}(a)$. For an alphabetised event (s, e) we want to record the dynamic interface s as well as the event e, thus enforcing ***MC1***. The value of s may be given by the value of the variable *mChans* at the time of the recording, thus enforcing ***MC4***. We apply ***MC2*** to constrain the values of *mChans*, *mChans*$'$, and ***MC3*** for *dref*$'$. The process that is ready to engage in event a and then increments its DAT when a has occurred, or simply records the current dynamic interface (to serve as the valid interface for the next process) is denoted by $do_{\Sigma}(a)$.

Definition 3.2.3 *($do_\Sigma(a)$) For any event $a \neq nil$:*

$$\begin{aligned} do_\Sigma(a) \;&\widehat{=}\; \mathbf{MC23} \circ \Phi(a \notin dref' \lhd wait' \rhd dtr' = dtr ^\frown \langle (mChans, a)\rangle) \\ \Phi \;&\widehat{=}\; \boldsymbol{R} \circ and_B = and_B \circ \boldsymbol{R} \end{aligned}$$

where $and_B(X) \;\widehat{=}\; B \wedge X,\; B \;\widehat{=}\; (tr' = tr \wedge wait') \vee tr < tr'$. □

Theorem 3.2.2 *Def. 3.2.3 obeys all the laws of prefix [Hoa85]–see also Appendix B.*

Proof $[\![\, do \,]\!]_{mob} = \mathbf{MC23} \circ \boldsymbol{MCr1} \circ \boldsymbol{MCr3} \circ \boldsymbol{MCcsp1}([\![\, do \,]\!]_{csp})$ is additive (cf. Thrm. 3.2.1). □

3.2.2. Channel-passing

Moving a channel has different effects depending on whether the channel is being moved out (released) or moved in (acquired).

Release. Moving out/sending out a channel implies that the channel must no longer be authorised, viz., it must be removed from *mChans*. Clearly, any attempt of moving out a channel, say *oldch*, not already owned must fail. The release event is recorded as in do_Σ, whence ***MC1*** holds. Because of **MC4**, the new value of *mChans* must be recorded into the trace, e.g., $\langle (mChans, rel.oldch)\rangle ^\frown \langle (mChans', nil)\rangle$. Then, any future refusal may not contain the event that has just been removed. This further means that all of the events related to the released channel must be removed from *dref* as well (if they were already in *dref*), to avoid chaotic behaviour—***MC3***. We apply ***MC2*** to constrain the values of *mChans*, *mChans'*.

Definition 3.2.4 **(Channel-passing output prefixes)** *Let oldch be the channel to be released; let α oldch denote the corresponding set of events. When the sending process moves out oldch and loses its value, we talk of* move *or* copy-then-delete *semantics:*

$$rel!!oldch \;\widehat{=}\; \boldsymbol{MC2} \circ \Phi \left(\begin{array}{l} rel.oldch \notin dref' \lhd wait' \rhd \\ \left(\begin{array}{l} dtr' = dtr ^\frown \langle (mChans, rel.oldch)\rangle ^\frown \langle (mChans', nil)\rangle \wedge \\ dref' = dref \backslash \alpha\, oldch \lhd (\alpha\, oldch \in dref) \rhd dref \wedge \\ mChans' = mChans \backslash \{oldch\} \end{array} \right) \\ \lhd oldch \in mChans \rhd \bot \end{array} \right)$$

When the sending process moves out oldch but still retains its value, we talk of copy *or* clone *semantics:*

$$rel!!_c oldch \;\widehat{=}\; \boldsymbol{MC2} \circ \Phi \left(\begin{array}{l} rel.oldch \notin dref' \lhd wait' \rhd \\ \left(\begin{array}{l} dtr' = dtr ^\frown \langle (mChans, rel.oldch)\rangle ^\frown \langle (mChans', nil)\rangle \wedge \\ mChans' = mChans \end{array} \right) \\ \lhd oldch \in mChans \rhd \bot \end{array} \right)$$ □

Acquisition. Moving in /receiving a channel *newch* requires that the receiving process must not own *newch* prior to receiving it. The value of *mChans* is incremented with *newch* and then recorded into the trace—for reasons invoked previously, the following definition is ***MC134*** healthy.

Definition 3.2.5 **(Channel-passing input prefix)**

$$acq??x \;\widehat{=}\; \boldsymbol{MC2} \circ \Phi \left(\begin{array}{l} \exists\, acq.newch \notin dref' \lhd wait' \rhd \\ \left(\begin{array}{l} dtr' = dtr ^\frown \langle (mChans, acq.newch)\rangle ^\frown \langle (mChans', nil)\rangle \wedge \\ mChans' = mChans \cup \{newch\} \wedge x' = newch \end{array} \right) \\ \lhd newch \notin mChans \rhd \bot \end{array} \right)$$ □

Theorem 3.2.3 *Defs. 3.2.5 and 3.2.4 obey all the laws of prefix [Hoa85]–see also Appendix B.*

Proof Defs. 3.2.5 and 3.2.4 are instances of prefix (Def. 3.2.3), thus they are additive definitions (cf. Thrm. 3.2.1). □

3.2.3. Parallel composition

The semantics of the parallel composition operator are similar to the static CSP definition, except that the trace merge must take into account the new structure of the events. The aim is to preserve the merge of events histories as defined in static CSP, and specify only that of the associated interfaces histories.

Channel-passing operations (cf. § 3.2.2) perform two actions that are somewhat causal: the input or output of a channel and then the record of the next interface. That is, the operations yield events of (loosely) the form $(i_0, mov.ch)\ \&(i_1, nil)$, viz., the events always go together and in that order. This ordering is guaranteed by the definitions of the channel passing operations, but may be broken in the presence of interleaving. Hence, any correct interleaving must preserve the expected ordering. For illustration, we would expect the following:

$$(i_0, mov.ch)\ \&\ (i_1, nil) \parallel (j, b) = \begin{pmatrix} \{\langle (i_0 \cup j, mov.ch)\ \&\ (i_1 \cup j, nil), (i_1 \cup j, b)\rangle\} \cup \\ \{\langle (i_0 \cup j, b), (i_0 \cup j, mov.ch)\ \&\ (i_1 \cup j, nil)\rangle\} \end{pmatrix}$$

The following definition also ensures that we never have sequences $(i, a)\ \&\ (i, nil)$, since (i, nil) elements mark the beginning of a new snapshot and should not occur after any other event within that snapshot.

Definition 3.2.6 (DAT parallel merge) *Let s and t be two traces. Let $E(s)$ denote the set of events in s. Let a, b, c, d be (pairwise distinct) events such that: $\{a, b\} \notin E(s) \cap E(t)$, $\{c, d\} \in E(s) \cap E(t)$. Then, we define the (DAT) trace merge for parallel composition by recursion as follows:*

$$\begin{aligned}
s \parallel t &\mathrel{\widehat{=}} t \parallel s \\
\langle (i, c)\rangle ^\frown x \parallel \langle (j, c)\rangle ^\frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, c)\rangle ^\frown u \mid u \in x \parallel y\} \\
\langle (i, c)\rangle ^\frown x \parallel \langle (j, d)\rangle ^\frown y &\mathrel{\widehat{=}} \{\} \\
\langle (i, a)\rangle ^\frown x \parallel \langle (j, c)\rangle ^\frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, a)\rangle ^\frown u \mid u \in x \parallel \langle (j, c)\rangle ^\frown y\} \\
\langle (i, a)\rangle ^\frown x \parallel \langle (j, b)\rangle ^\frown y &\mathrel{\widehat{=}} \begin{pmatrix} \{\langle (i \cup j, a)\rangle ^\frown u \mid u \in x \parallel \langle (j, b)\rangle ^\frown y\} \cup \\ \{\langle (i \cup j, b)\rangle ^\frown u \mid u \in \langle (i, a)\rangle ^\frown x \parallel y\} \end{pmatrix} \\
\langle (i, nil)\rangle ^\frown x \parallel \langle (j, c)\rangle ^\frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, nil)\rangle ^\frown u \mid u \in x \parallel \langle (j, c)\rangle ^\frown y\} \\
\langle (i, nil)\rangle ^\frown x \parallel \langle (j, nil)\rangle ^\frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, nil)\rangle ^\frown u \mid u \in x \parallel y\} \\
\langle (i, nil)\rangle ^\frown x \parallel \langle (j, a)\rangle ^\frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, nil)\rangle ^\frown u \mid u \in x \parallel \langle (j, a)\rangle ^\frown y\}
\end{aligned}$$ □

Definition 3.2.7 (Parallel composition)

$$\begin{aligned}
P \parallel Q &\mathrel{\widehat{=}} P(\mathbf{o}, 1.\mathbf{o}') \wedge Q(\mathbf{o}, 2.\mathbf{o}') \mathbin{;} M(1.\mathbf{o}, 2.\mathbf{o}, \mathbf{o}') \\
M &\mathrel{\widehat{=}} \begin{pmatrix} ok' = (1.ok \wedge 2.ok) \wedge \\ wait' = (1.wait \vee 2.wait) \wedge \\ (dtr' - dtr) = (1.dtr - dtr) \parallel (2.dtr - dtr) \wedge \\ dref' = 1.dref \cup 2.dref \wedge \\ mChans' = 1.mChans \cup 2.mChans \end{pmatrix} \mathbin{;} SKIP
\end{aligned}$$ □

Theorem 3.2.4 *Def. 3.2.7 obeys all parallel composition laws [Hoa85]–see also Appendix B.*

Proof Recall, $mtr = \pi_1(dtr)$, $tr = \pi_2(dtr)$. Then
$[\![M]\!]_{mob} = [\![M]\!]_{csp} \wedge (mtr' = 1.mtr \parallel 2.mtr \wedge mChans' = 1.mChans \cup 2.mChans)$
is additive (cf. Thrm. 3.2.1). □

Since a process cannot create a channel by itself, the interface of a parallel composition cannot grow of its own.

3.2.4. Renaming

The interaction of renaming (resp. hiding) and channel mobility gives rise to a number of issues, e.g., it should not be possible to rename (resp. hide) *in advance* a name not yet acquired. This section is concerned with the semantics of renaming. The semantics of hiding will be presented in Sect. 3.2.5.

Formalisation and semantics

Let $P = a \rightarrow SKIP$. $P[b \leftarrow a] = b \rightarrow SKIP$ is the process that engages in action b whenever P engages in action a. Let $f : \mathcal{I}\,P \rightarrow \mathcal{I}\,f(P)$. Then (in static CSP): $f(P) = P[\mathcal{I}\,f(P) \leftarrow \mathcal{I}\,P]$. Renaming applies to channels as well, when we would like to define the process $P[ch_2 \leftarrow ch_1]$ that engages in a given channel ch_2 whenever P engages in a distinct channel ch_1. Let $P = ch_1!y \rightarrow SKIP$. Then $P[ch_2 \leftarrow ch_1] = ch_2!y \rightarrow SKIP$. Renaming works here since $ch_1 \in \mathcal{I}P$. In contrast, $P[ch_2 \leftarrow ch_3] = P$ is vacuous since $ch_3 \notin \mathcal{I}P$. A number of issues arise, however, in the presence of channel mobility.

Let $Q = in??x \rightarrow x!v \rightarrow SKIP$. **Name mismatch** occurs when one tries the renaming $Q[ch_2 \leftarrow ch_1]$, in anticipation that $x = ch_1$ (foresight). If Q never inputs ch_1 then the intended renaming is vacuous, otherwise, it would be unhealthy for two reasons: (1) **unknown names**, which occur in applications where it is actually impossible of knowing in advance what channels will be acquired in the future; nonetheless, since renaming must defined over $\mathbb{P}\,MCh$, (2) **name collision** can occur if a substitue name coincides with an acquired name. E.g., let $R = ch_1!w \rightarrow Q$ and assume $x = ch_2$, then $R[ch_2 \leftarrow ch_1] = ch_1!w \rightarrow \bot$ is unhealthy. This indicates that channel mobility provides a scope or limit to the application of renaming such that renaming should not apply to acquired names.

Static renaming

Let $mChans^k$ ($k \in \mathbb{N}$) denote the k-th value of $mChans$ recorded in a given trace (Def. 3.1.8). If, for all k, $mChans^k = mChans^{k+1}$, then, the corresponding process is static and none of the issues mentioned above can occur. The semantics of renaming is thus strictly identical to the definition in static CSP. Else, if there exists a k, $mChans^k \neq mChans^{k+1}$, then, we are in the presence of channel mobility and we must deal with the issues mentioned previously.

Syntactically, the only operation that implies $mChans^k \neq mChans^{k+1}$ and which should concern us is channel passing input prefix—channel passing output prefix would render renaming vacuous for the removed channel. We can then avoid name mismatch or foresight by using the following definition:

$$\begin{aligned} f(in??x) &= f(\Box_{newch}\, in.newch) \,⨾\, f(x := newch) && (\textbf{no}-\textbf{foresight}) \\ &= \Box_{f(newch)} f(in.newch) \,⨾\, x := f(newch) \\ &= \Box_{newch} f(in).newch \,⨾\, x := newch \end{aligned}$$

(**no-foresight**) guarantees the following postcondition for the renaming operation: *if* $f(mChans^0)$ *is defined (viz.* f *is defined over channels in* $mChans^0$*) then for all* $k > 0$, $f(mChans^{k+1} - mChans^k) = mChans^{k+1} - mChans^k$ *(viz.* f *should be undefined over channels in* $mChans^{k+1} - mChans^k$*; however, because* f *must be injective to avoid undefinedness,* f *behaves like the identity function over future names).*

We must further ensure that if $ch \in mChans^0$ and $ch \in (mChans^{k+1} - mChans^k)$, then $f(ch)$ is defined/renamed upto the snapshot $[k, k+1]$, from which $f(ch) = ch$. That is, channel mobility limits the scope of the application of renaming. (**no-foresight**) does not guarantee scope restriction, because it is based on syntax. We must impose (renaming) scope restriction on the semantics of renaming, viz.,

$$\begin{aligned} f(P) \Rightarrow \forall\, ch, \forall\, k \geq 0 \bullet (ch \in mChans^0 \wedge ch \mapsto f(ch)) \vee && (\textbf{rename}-\textbf{scope}) \\ ch \in mChans^0 \cap (mChans^{k+1} - mChans^k) \wedge ch \mapsto f(ch) = ch \end{aligned}$$

rename−scope condition is also sufficient to ensure that renaming has no effect on mobility.

Let Amc denote the set of all acquired mobile channels during a given observation. Then, $mChans^0 - Amc$ denotes initial channels that are not later released-then-acquired. These are the only channels that can be renamed.

Definition 3.2.8 (Acquired mobile channels) *Let $E(t)$ denote the set of all the elements of a given trace t, e.g. $E(\langle ..., (s_k, e_k), ...\rangle)$ is composed of elements (s_k, e_k).*

$$\begin{aligned} Amc &\mathrel{\widehat{=}} \bigcup_{0 \le k \le \#\, dtr'} (mChans^{k+1} - mChans^k) \qquad \square \\ &= \bigcup_{0 \le k \le \#\, dtr'} \{s_{k+1} - s_k \mid s_k \in E(\pi_1(dtr'))\} \end{aligned}$$

As a consequence, **rename−scope** condition can be simplified to: $f(P) = f\mid_{mChans^0 - Amc}(P)$.

We now give the definition of static renaming in the presence of channel mobility. In static CSP, a renaming function f is defined over a constant set, the interface $\mathcal{I}$. In mobile CSP, it must be defined over the variable set $mChans$ such as to obey **rename−scope**. We thus define renaming in terms of the substitution operation (cf. Appendix B) as follows:

Definition 3.2.9 (Static renaming) *Let $u = \{dtr, dref, mChans\}$. Let $f : \mathbb{P}\, MCh \to \mathbb{P}\, MCh$ be an injective function. Let $f\mid_B : B \to \mathbb{P}\, MCh$ denote the restriction of f on a given (non-empty) set $B \in \mathbb{P}\, MCh$.*

$$f(P) \mathrel{\widehat{=}} P[f(u)/u, f(u')/u'] \qquad \textbf{\textit{where}} \quad f(dtr' - dtr) = f\mid_{mChans^0 - Amc}(dtr' - dtr) \qquad \square$$

Theorem 3.2.5 *Def. 3.2.9 obeys all the laws of renaming [Hoa85]–see also Appendix B.*

Proof Def. 3.2.9 is just a reformulation of (static CSP renaming) to enforce **rename−scope**, which trivially holds in static CSP since $\forall\, k \bullet mChans^{k+1} = mChans^k$. $\square$

Notation. Subsequently, for convenience, the notation $[ch]$ will be used denote the variable that has (presumably received) channel ch for value.

Example 3.2.1 *Let $P = ch_1!x \to acq??[ch_2] \to [ch_2]!y \to SKIP$.*

1. *$P[ch_2 \leftarrow ch_1] = ch_2!x \to \bot$ is* erroneous *since ch_2 is not known before its acquisition.*
2. *$P[ch_3 \leftarrow ch_2] = P$ is* vacuous *since $ch_2 \notin mChans^0$.*
3. *$P[ch_3 \leftarrow ch_1] = ch_3!x \to acq??[ch_2] \to [ch_2]!y \to SKIP$ is a* valid *(static) renaming.* $\square$

Example 3.2.2 (Renaming scope restriction) *Let $Q = ch_1!x \to rel!!ch_1 \to acq??[ch_1] \to [ch_1]!y \to SKIP$. Then, $Q[ch_2 \leftarrow ch_1] = ch_2!x \to rel!!ch_2 \to acq??[ch_1] \to [ch_1]!x \to SKIP$ is a* valid *(static) renaming.*

More generally, f is a valid static renaming function iff

$$f(ch.v \to rel!!ch \to acq??[ch] \to [ch].v \to SKIP) = f(ch).v \to rel!!f(ch) \to acq??[ch] \to [ch].v \to SKIP$$

$\square$

3.2.5. Hiding

Formalisation

A process does not need to know at all whether or not a channel is silent (i.e., hidden from the environment, does not appear in the interface). Indeed, the *knowledge of a channel* confers a communication functionality (over the given channel), not an abstraction functionality. It is *hiding* (the operator) that provides the abstraction functionality. Thus, communication concerns should be separated from abstraction concerns. In other words, for a given process P, the specification of its interface $\mathcal{I}P$ only signifies that P may communicate through a given channel $ch \in \mathcal{I}P$. It does not matter how and when and if ch was acquired at run-time. It is hiding, $P \setminus X$, that specifies in X what channel is to be considered silent.

Channel mobility opens up the possibility for hiding acquired channels at run-time. The corresponding operation will be called *dynamic hiding*, is common in both telecommunications network and computer networks. For example, some channels can be leased for private use by a network operator, thus forming a Private Branch eXchange (PBX). Other users in the network cannot access the PBX, and many PBXs can be created throughout the lifetime of the network, thus constituting instances of dynamic hiding. Dynamic hiding can also occur in a client-server application when, for example, a new server is added into the network for load balancing. The link between the old server and the new one is generally invisible to the clients, thus constituting an instance of dynamic hiding. In this paper, we will limit ourselves to the semantics of hiding as defined in static CSP, henceforth called static hiding. Dynamic hiding requires first extending (mobile) UTP-CSP theory, which is beyond the scope of this paper.

Channel mobility initroduces the possibility of *name collision* between a newly acquired channel and a silent one. This is unhealthy. As for renaming earlier, channel mobility restricts the scope of application of hiding such that hiding should not apply to acquired names.

Static hiding

In static CSP, hiding applies to the channels in the interface only, viz., $P \setminus X = P \setminus X \cap \mathcal{I}P$. In mobile CSP, we must ensure that hiding applies to channels in the initial interface only. Since in mobile CSP a channel in the initial interface may be released then acquired, hiding must not apply to an acquired name, independently of its release/acquisition history. That is, only channels in $X \cap (mChans^0 - Amc)$ can be hidden.

Definition 3.2.10 (Static hiding)

$$P \setminus X \mathrel{\widehat{=}} \exists\, tra, refa \bullet \begin{pmatrix} P[tra/dtr', refa/dref', mChansA/mChans, mChansB/mChans'] \wedge \\ tra = dtr' \upharpoonright (mChans^0 \cup Amc) - (X \cap (mChans^0 - Amc)) \wedge \\ refa = dref' - (X \cap (mChans^0 - Amc)) \wedge \\ mChansA = mChans - (X \cap (mChans^0 - Amc)) \wedge \\ mChansB = mChans' - (X \cap (mChans^0 - Amc)) \end{pmatrix} \mathbin{;} SKIP \qquad \square$$

Theorem 3.2.6 *Def. 3.2.10 obeys all the laws of (static) hiding [Hoa85, §3.5]–see also Appendix B.*

Proof Def. 3.2.10 is just a reformulation of (static CSP hiding)—cf. Appendix B, where $X = X \cap \mathcal{I}P$ is replaced by $X \cap (mChans^0 - Amc)$, since the interface is now variable or dynamic. □

Example 3.2.3 (Hiding scope restriction)

$$(ch.v \to out!!ch \to in??[ch] \to [ch].v \to SKIP) \setminus \{ch\} = out!!ch \to in??[ch] \to [ch].v \to SKIP$$

Notice the similarity with the static CSP process $(a \to SKIP) \setminus \{a\} \mathbin{;} a \to P$. In fact, we have an equivalence if we also hide the channel passing actions as follows:

$$(ch.v \to out!!ch \to in??[ch] \to [ch].v \to SKIP) \setminus \{ch, out, in\} = [ch].v \to SKIP \qquad \square$$

3.3. Example: a mobile telecommunications network

A mobile telecommunications network has three main elements: a user/client (caller or receiver); a base station or BTS and a control station or BSC—see Fig. 2.

- Client: dials a number and then waits for the acquisition of a (new) transmission channel; then, it engages in a conversation for some time. At the end of the conversation, it releases the channel acquired previously; during the conversation, it may receive a handover request (through signalling channels) in which case it releases its current transmission channel and waits for the acquisition of a new one. Note that we are interested in the *hard handover* in which the client can hold a single transmission line only.

$$\begin{aligned} Client &\mathrel{\widehat{=}} dial \to alloc??[talk, listen] \to Chat \\ Chat &\mathrel{\widehat{=}} ([talk]!msg \mathbin{\Box} [listen]?msg) \to \big((Chat \sqcap Hangup) \mathbin{\Box} Handoff\big) \\ Hangup &\mathrel{\widehat{=}} hangup \to release!![talk, listen] \to Client \\ Handoff &\mathrel{\widehat{=}} handoff \to release!![talk, listen] \to alloc??[talk, listen] \to Chat \end{aligned}$$

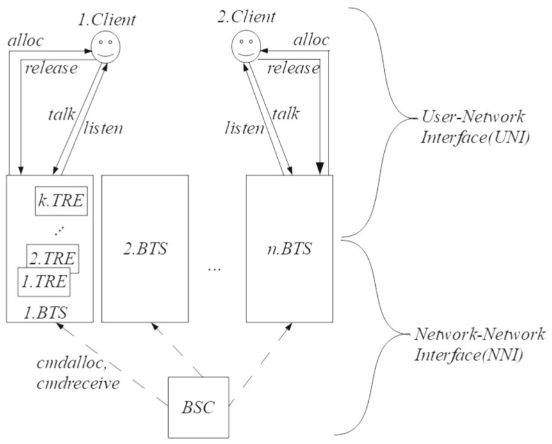

Fig. 2. A mobile telecommunications network

talk and *listen* are the channel ends received by a client, *Client*, through the channel carrier *alloc*. The event *hangup* models the action of a client that puts an end to a transmission—this is the normal termination of a call. Then, the client releases its *talk* and *listen* channels through the channel carrier *release* and is again ready to dial a number. A client also releases its *talk*/*listen* channels when it receives a handoff request, modelled by the *handoff* event; this time however, the client waits for another *talk*/*listen* channels before resuming its conversation.

- Base Station (or Base Transceiver Station or BTS): is composed of a limited number radio (or transmitter/receiver or TRE) stations and is in slave-master relation with a BSC. Let $maxTRE \geq 2$ denote the maximum number of TRE that a BTS can contain. Each TRE is either idle, then it is waiting for a communication with some Client, or it is transmitting, or it is faulty. A faulty TRE can no longer communicate; it is defined as the process that has disposed of its channel (ends).

$$\begin{aligned}
BTS &\;\widehat{=}\; (\;\big\|_{1 \leq k \leq maxTRE}\, k.TRE) \parallel BSCslave \\
TRE &\;\widehat{=}\; (listen?msg \rightarrow talk!msg \rightarrow TRE) \sqcap FltyTRE \setminus \{dispose\} \\
FltyTRE &\;\widehat{=}\; (dispose!!talk \sqcap dispose!!listen) \rightarrow SKIP \\
BSCslave &\;\widehat{=}\; cmdallocate \rightarrow alloc!![talk, listen] \rightarrow (Retrieve \;\Box\; cmdretrieve \rightarrow Retrieve) \\
Retrieve &\;\widehat{=}\; (retrieve??[talk, listen] \rightarrow BSCslave)
\end{aligned}$$

TRE receives messages from a client and forwards them to another. Above, a fault occurs randomly, then *TRE* behaves like *FltyTRE*. *FltyTRE* disposes of the faulty channel through the *dispose* channel and then does nothing. The definition of *TRE* may be refined by using the interrupt operator $\triangle_{iev}$ [McEw10] instead. Then, it is the ocurrence of an interrupt event *iev* that would cause *TRE* to behave like *FltyTRE*. Notice the use of $listen?x$ (resp. $talk!y$), instead of $[listen]?x$ (resp. $[talk]!y$), since *talk* and *listen* channels belong to the BTS already, whilst a client needs to acquire them instead.

BCSslave receives commands from the BSC, then it releases its communication links with a Client. *cmdallocate* makes the BTS allocate a channel to a client; *cmdretrieve* makes the BTS retrieve a channel from a client. When the client hangs up, the BTS automatically retrieves its communication links, without

prior receiving a command *cmdretrieve* from a BSC. *retrieve* is the channel carrier used by the BTS for retrieving a transmission channel that was previously sent to a client; it is also the counterpart of the *release* channel used by the client when releasing an acquired channel.
- Control Station (or Base Controller Station or BSC): may communicate with a limited number of assigned BTSs through signalling/command links, *cmdallocate*, *cmdretrieve*. Let $maxBTS \geq 2$ denote the maximum number of BTSs that can be supervised by a single BSC.

$$\begin{aligned}
BSC &\mathrel{\widehat{=}} \bigsqcap_{j<maxBTS} j.cmdallocate \rightarrow (BSC \sqcap HandOver \sqcap LoadB \sqcap LoadB2) \\
HandOver &\mathrel{\widehat{=}} handoff \rightarrow cmdretrieve \rightarrow BSC \\
LoadB &\mathrel{\widehat{=}} loadb \rightarrow HandOver \\
LoadB2 &\mathrel{\widehat{=}} loadb \rightarrow BSC
\end{aligned}$$

A handover operation occurs on two instances. In the first instance, the handover is due to a client changing its coverage area: this is modelled by the process *HandOver* (viz. *handoff* event). In the second instance, the handover is due to load balancing (the client has not changed its coverage area, but the latter is full): this is modelled by the process *LoadB* (viz., *loadb* event). Load balancing may occur before the attribution of channels to a client, when the client is in an area that was full before the client initiated its call: this is modelled by process *LoadB*2. After either a *handoff* or a *loadb* event, the *BSC* commands the current BTS to release its communication links and another BTS to communicate to the client.
- User-Network Interface (or UNI): the part of a telecommunications network composed principally of clients and BTSs. Below we define a subnetwork covered by a single BSC. Let BTS^n denote a collection of a given number n of BTS.

$$\begin{aligned}
UNI &\mathrel{\widehat{=}} ClientBase \parallel BTS^n \parallel BSC \\
ClientBase &\mathrel{\widehat{=}} \Big\|_{i \in I} i.Client && \text{where } I \subseteq \mathbb{N} \setminus \{0,1\} \\
BTS^n &\mathrel{\widehat{=}} \Big\|_{2 \leq j \leq n} j.BTS && \text{where } n \leq maxBTS
\end{aligned}$$

Note that when a channel fault occurs, say on the nth BTS,

$$BTS^n = (\Big\|_{2 \leq j \leq n-1} j.BTS) \parallel SKIP = BTS^{n-1}$$

- In the previous definitions, the BTS had a static number of TREs which could decrease in case of a faulty channel. In what follows we describe the case where the BTS may acquire new TREs. Let BTS_k denote a BTS containing a given number k of TRE.

$$\begin{aligned}
DynaBTS &\mathrel{\widehat{=}} \Box_{2 \leq k \leq maxTRE} BTS_k \\
BTS_{k+1} &\mathrel{\widehat{=}} BTS_k \parallel newTRE \\
newTRE &\mathrel{\widehat{=}} new??[talk, listen] \rightarrow (k+1).TRE \lhd k < maxTRE \rhd SKIP
\end{aligned}$$ □

4. Further considerations

Interrupt. The interrupt operator, $P \bigtriangleup Q$, models the behaviour of a process P whose execution may be interrupted by a process Q. An interrupt event, denoted by i, is then used to model the occurrence of an interrupt. In UTP, McEwan et al. [McEw10] define semantics of the interrupt operator (for reactive processes) based on sequential composition. That is, in modified $P \,;\, (i \rightarrow Q) = P \bigtriangleup_i Q$, Q is allowed to start executing if P is in a waiting state; whilst in traditional sequential composition, P must terminate before Q is allowed to start. Different semantics may be given to the interrupt. In particular, the interrupt operator can be defined based on parallel composition. Then, explicit representation of time is often used. In modified $P \parallel i \rightarrow Q = P \triangle i \rightarrow Q$, an analysis of the time stamps of the events from P and the occurrence of the interrupt event i allows recovering the final trace as follows: $\langle a * t_0, a * t_1, ..., a * t_{k-1}, i * t_k, b * t_{k+1}, ...\rangle$, where $a \in \mathcal{A}P$, $i, b \in \mathcal{A}Q$.[4] From what precedes, it comes that the occurrence of an interrupt event has of itself no effect on channel mobility: the interrupt will occur either before the channel-passing action or after.

There is a case, however, where interrupt can have an interesting effect on channel mobility: when process migration or code mobility occurs. Process migration [Fug98] can be defined as the interruption of an executing unit (process, source), followed by its migration to a target executing environment where the interrupted process will resume its execution. In other words, process migration is "interrupt-then-code mobility". The author has provided a formal semantics for process migration as a form of interrupt mechanism where the interrupt routine, i.e., Q in $P \triangle Q$, performs code mobility (cf. [Ek16, §5.5]). Then, if the code to be moved contains a channel-passing operation, the effect of channel-passing will likely be observable in the target process instead of the source. Tang and Woodcock [Tang04b] provide semantics for code mobility in UTP-CSP. We note here that modelling process migration requires modelling unstructured programming primitives such as jumps. The author and Woodcock have used the concept of *continuation* to represent jumps in UTP, [Ek16b], extending prior work from Hoare and He [HoaHe98, Ch. 6]. Thus, a theory that aims to account for both channel mobility and process migration will be somewhat complex, as it should account for continuations, time, un-timed message-passing and channel-passing features.

Refinement. Let S and D denote any static and dynamic processes, respectively. Let *snap* denote the number of snapshots of a process. Then, $\mathit{snap}(S) = 1$ and $\mathit{snap}(D) \geq 2$. Let $D = S_1 \mathbin{;} \mathit{car}.\mathit{unkn} \mathbin{;} S_2$. If we view *unkn* to be a symbolic value, viz., a channel itself, then, the interface of D is computable at compile time, i.e., $\mathcal{I}D = \mathcal{I}S_1 \cup \{\mathit{car}\} \cup \mathcal{I}S_2 \cup \{\mathit{unkn}\}$.[5] As a consequence, we can reason about the traces and refusals of D as in static CSP; hence, the refinement calculus ([Ros10], cf. Appendix D) remains unchanged. However, *unkn* is actually a variable, and mappings $\mathit{unkn} \mapsto \mathit{new}$ are important for refinement.

Let $D' = S_1' \mathbin{;} \mathit{car}.\mathit{unkn} \mathbin{;} S_2'$. We want to establish the conditions for refinement between D and D'. For ease, we will write $A <> B$ to mean that sets A and B are disjoint, i.e., none is the subset or equal of the other: $\neg\,(A \subseteq B \vee B \subseteq A)$. Clearly, when the *initial* interfaces of D and D' are disjoint ($\mathcal{I}S_1' <> \mathcal{I}S_1$), refinement cannot be established between them. Also, if $\mathcal{I}D \subseteq \mathcal{I}D'$ up to a given snapshot and later $\mathcal{I}D \supseteq \mathcal{I}D'$, refinement cannot be established.

Let $D.ch$ denote channel $ch \in \mathcal{I}D$. Let $\mathcal{I}S_1' \subseteq \mathcal{I}S_1$ and let $\mathcal{I}S_2 \backslash \{S_2.\mathit{unkn}\} \subseteq \mathcal{I}S_2' \backslash \{S_2'.\mathit{unkn}\}$. Then, were it not for channel passing input, D and D' would be in a refinement relation. If $D.\mathit{unkn} \neq D'.\mathit{unkn}$, then $\mathcal{I}S_2 \cup \{D.\mathit{unkn}\} \neq \mathcal{I}S_2' \cup \{D'.\mathit{unkn}\}$ and there can be no refinement between D and D'. However, if $D.\mathit{unkn} = D'.\mathit{unkn} = \mathit{kn}$, then $\mathcal{I}D = \mathcal{I}S_1 \cup \{\mathit{car}\} \cup \mathcal{I}S_2 \cup \{\mathit{kn}\} \supseteq \mathcal{I}D' = \mathcal{I}S_1' \cup \{\mathit{car}\} \cup \mathcal{I}S_2' \cup \{\mathit{kn}\}$. Thus, the refinement calculus remains unchanged.

In order to formally achieve the mapping $\mathit{unkn} \mapsto \mathit{kn}$ mentioned above, we remark that if D is sent a channel kn, then D will receive kn. Then, we need only to place D and D' in an environment, say $\mathcal{E}$, that has kn in its interface. Then, $D \sqsubseteq D' \Leftrightarrow \forall \mathcal{E} : D \parallel \mathcal{E} \sqsubseteq D' \parallel \mathcal{E}$. It remains for us now to formalise the conditions expressed above in terms of traces and refusals. We leave this for future work.

5. Related work

The concept of a DAT (Dynamic Alphabetised Trace) captures changes of interface, and similar dynamic traces have been used in [HoaOh08]. However, we have seen that without the concept of capability, DATs are insufficient for characterising channel mobility. Hence, mobile processes must have a static capability and a dynamic interface. The notion of capability is original to this work. Its introduction to the works cited below is necessary for their validity and would pose no difficulty.

No model in the literature is a direct extension of CSP as in this work. Vajar et al. [Vaj09] propose an extension of CSP||B with channel mobility. [Vaj09] explores a limited form of mobility where channels

[4]Obviously, the following traces are also possible: $\langle a * t_0, a * t_1, ..., a * t_k\rangle$ when no interrupt occurs; $\langle i * t_0, b * t_1, ..., b * t_k\rangle$ when interrupt occurs before P starts.

[5]Recall $\mathcal{I}P$ denotes the interface of a (static) process P. For readability, here, we override $\mathcal{I}$ to mean also the interface of a mobile process, mChans. Then, $\mathcal{I}_0$ denotes the initial interface of a process, i.e., mChans^0.

are passed from B machines to CSP controllers only. There is no concept of capability, and the interface of processes is not discussed. As a consequence, no healthiness condition is defined over traces of mobile processes.

Hoare and O'Hearn [HoaOh08], and Roscoe [Ros10a, Ros10b, Ros10] both propose preliminary models for mobility in CSP. Hoare and O'Hearn [HoaOh08] investigate an analogy between separation logic and channel mobility from which they extend CSP. [HoaOh08] contains a different traces model than the DATs used here. Indeed, instead of pairs (interface, event), each interface and event occurrence is recorded separately. Then, in a trace, odd elements are all interface values whilst even elements are events. There is no concept of capability, and their investigation is limited to traces only, whilst our work covers also failures and divergences. In [HoaOh08], a channel can transmit a copy of itself, which to us is very counter-intuitive.

In [Ros10a, Ros10b], Roscoe proposes different possible models for mobile CSP, all based on CSP *operational semantics*. Unlike [Vaj09] and [HoaOh08] whose approach may be qualified as 'dynamic traces', [Ros10a, Ros10b] attempt to model Pi-calculus [Mil99] fresh names mechanism in CSP. This results in many models with complex semantics. The models are also limited to traces semantics, although directions to extend that work to failures are given. In [Ros10], Roscoe introduces a closed-world semantics for mobile CSP, but the concept of closed-world is not well-defined therein. Indeed, if $P \parallel Q$ is closed-world [Ros10] means that P and Q alone can exchange channels, then P and Q individually must be open-world. However, in [Ros10], Roscoe proposes closed-world as the basis for defining open-world semantics.

Bialkiewicz and Peschanski [BialPes09b] provide a CSP-like traces model for Pi-calculus [Mil99]. The resulting traces model, which we call *localised traces*, takes into account both fresh names—constructed differently from [Ros10b]—and branching information. The localised traces model is significantly more complex than traditional CSP traces and, since it is targeted to represent Pi-calculus processes, it is not evident how to relate it to CSP itself.

Welch and Barnes [WelBa08] propose a CSP semantics for the programming language occam-pi, however, their result is not abstract enough. We discuss how to model occam-pi channel mobility mechanism in greater detail in [Ek16, §3.2.2]. Hereafter is a summary of our modelling approach. Occam-pi is a process-oriented programming language that supports channel mobility, amongst other features. A channel is represented as a pair of channel ends, and all the channels that can be used in the communication between two processes are grouped together as a *bundle*. A bundle has two ends called client and server respectively. Both bundles and channels are defined/implemented as *mobile data types*. In [WelBa08], bundles and mobile channels are modelled as *indexed* CSP processes, respectively, $Bdle(bId, nbFlds)$ and $ChFld(bId, chId)$, where bId uniquely identifies a CSP bundle process, $nbFlds$ its number of channel-field (component) processes; $chId$ uniquely identifies a channel-field process within a bundle. Essentially, $Bdle(bId, nbFlds) = \parallel_{1 \leq chId \leq nbFlds} ChFld(bId, chId)$. $ChFld$ is a process that communicates through three channels: rd for input, wr for output, and ack for acknowledgement. An occam-pi program, $O\pi$, is modelled as the parallel composition of two processes: the application (system) process, AS, and the mobile channel kernel, MCK, i.e., $O\pi = AS \parallel MCK$. AS is the actual application, and MCK is a server of channels. AS processes can communicate through static channels as typical CSP processes do; or else they can communicate through bundles provided by MCK. An AS process may request MCK to create a new bundle, then will receive bId in return. Channel mobility is then achieved by communicating bundle indexes from one AS process to another.

In [WelBa08], since channels are modelled as processes, one may rightly expect to see (bundle and channel-field) processes move—this is not the case. The model closely follows the implementation architecture, hence it is not abstract enough. Let $AS = P \parallel Q$. Since in [WelBa08] the interfaces of processes do not and cannot change—given that indexes are passed around and not channels themselves—we are bound to conclude that the passing of bId say from P to Q is like a guard on the use of $ChFld$ channels, as Q would need to have $ChFld$ channels in its interface already, by definition—occam-pi programs are static CSP processes, in [WelBa08]. We can simplify the model from [WelBa08] if we remark that $ChFld$ channels only are indexed, i.e., by force of the indexing mechanism, $ChFld$ channels become triples $(bId, chId, chan)$, where $chan \in \{rd, wr, ack\}$. An indexed channel is simply a channel. (Since CSP channels are unique by their name, providing uniqueness by indexing is superfluous.) We also amalgamate all three rd, wr, ack channels into a single mobile (CSP) channel, say mch. We are thus left with $Bdle = ChFld$, with interface $\{mch\}$. Then, we eliminate the process $Bdle$ by reducing it to its interface, given that it served only to provide channels to AS. Similarly, we eliminate MCK by reducing it to its interface, which is the set of all possible mobile channels. Given that any AS process could access MCK process, we add MCK's interface to the definition of all AS processes—this corresponds to the capability of AS, not its interface. We are thus left with a single process, AS (viz., $O\pi = AS$), whose capability is defined by the interface of MCK; the interface of AS is determined by a set of static channels and the interface of $Bdle$ processes (viz., the set of indexed channels—varies with the index acquired/released by the AS process), which defines mobile channels. From this, we easily achieve channel mobility by modelling every index passing operations say $move!bid$ by a channel passing operation $move!!mch$, where pairs (bid, mch) are unique.

In the context of dataflow models, Grosu et al. [GroSto99, Sto99] provide an extension of FOCUS with channel mobility, called mobile FOCUS. This is the first attempt known to the author for giving denotational semantics to mobility. The semantics of a FOCUS component are given by its messages history, extracted from its communications history. In mobile FOCUS, channels are moved as messages between components (or processes, in CSP terms), whose semantics are defined by timed histories. Time is central in characterising mobility as it is used to restrict the valid channels at a given observation time. Furthermore, [GroSto99] considers that time is *essential* for characterising channel mobility, that is, channel mobility could not have been characterised without time. However, the results presented in this paper contradict the later conclusion. In effect, channel mobility implies a time division between the time before and the time after the movement. That division is implemented in the difference between the interface before and the interface after (the movement), so that it is not necessary to represent time explicitly in their model.

Stølen [Sto99] considers the environment of a FOCUS component as any other component, thus channels are hidden from the global interface, as in static hiding. However, [Sto99] distinguishes for each component its internal interface from its external interface, and then imposes that each component should have a disjoint internal interface. Whilst the external interface can grow, the internal interface remains static. Hence, the separation between internal and external interface only serves to prevent *name collision*, such that an internal name may not appear in the trace.

Recently, Woodcock et al. [Wood15] present a version of mobile CSP based on the Reactive Designs formulation of UTP-CSP [CavWoo06]. In contrast, this paper contains a Reactive Processes formulation of UTP-CSP. [Wood15] provides interesting examples for the application of mobility in general, and mobile CSP in particular. Renaming and hiding notably are not discussed in [Wood15]. It would be interesting to study how channel mobility affects the link [CavWoo06, §6.2, Thrm.2, p.38] between Reactive Processes and Reactive Designs.

In the literature of Pi-calculus, no work provides semantics for the renaming operator.

In order to model secrecy, Giunti et al. [Giu12] define a hiding/secrecy operator for Pi-calculus, resulting in a new theory called Secret Pi-calculus. In [Giu12], the main effect of the secrecy operator is to prevent the extrusion or output of (channel) names declared secret. The secrecy operator, therefore, must not be confused with CSP hiding.

In [Sang96], Sangiorgi et al. investigate the possible relationship between CCS and Pi-calculus. They distinguish *internal* from *external mobility*, leading them to define a specific characterisation of internal mobility for Pi-calculus, called πI (see also [Borea98]). Internal mobility applies to Pi-calculus with a leftmost restriction operator (viz. $\nu x(P \parallel Q)$ or $(\nu xP \parallel \nu yQ)$, where P and Q are Pi-calculus processes)—only restricted names, x, y can be moved. Their treatment of external mobility yields more complex formulations than πI. In external mobility, non-restricted names can also be moved. The main—perhaps the only—difference between internal and external mobility is that the latter allows the possibility for a process to acquire a name that it already owned. Hence, our work models internal mobility. Furthermore, we consider internal mobility to be the only valid definition of channel mobility. It is nonetheless possible to remove the restriction on acquired names in our semantics simply by removing the corresponding condition, $ch \in mChans$, from the semantics of channel passing input prefix (cf. Def. 3.2.5).

6. Conclusion

We have presented an extension of static CSP with channel mobility, based on UTP semantics framework. The resulting theory, called mobile CSP, allows passing channels as messages between processes as in Pi-calculus. Mobile processes are charaterised as having a static capability (viz., introduced to model the knowledge, by a process, of the existence of channels) and a dynamic interface. New operations have been defined for channel passing, namely channel passing input and output prefixes. A process cannot receive a channel that it already owns, otherwise channel mobility would be vacuous; and it cannot send a channel it does not own already. Channels are characterised as concrete entities. Of particular importance is the fact that a channel name does not entirely characterise a channel, but its presence in the interface does.

We have shown that mobility must be defined from the perspective of a process, for which the world is always open, as modelled by the capability set. This perspective being the right choice is confirmed when dealing with renaming and hiding. We thus avoided a number of difficulties encountered in [GroSto99] and [Ros10b]. Overall, this work has resolved a number of issues related to representation and manipulation of (channel) names in calculi of mobility, notably issues of closed- and open-world, of internal and external interface, issues that may greatly cloud understanding and reasoning.

Two operators of interest to the author are dynamic renaming, which would allow a process to rename acquired names at runtime, and dynamic hiding, which would allow a process to hide acquired names at runtime. The author expects to present these operators in the near future. The mechanisation of mobile CSP is an interesting future work that would confirm the preservation of the laws of operators and further allow analysing/proving properties about mobile systems in a theorem prover.

Whilst Pi-calculus remains the main reference for formal semantics of channel mobility, we think that mobile CSP provides a more solid semantic basis. We expect future applications of mobile CSP to prove this claim to be valid. For example, a result available in the CSP world but not in Pi-calculus is the definition of transformations from $\boldsymbol{DN}$ healthy processes into $\boldsymbol{SN}$ processes. The latter result will soon be published.

A formal comparison with Pi-calculus is still missing, although attempts have been made to bridge the gap between (static) CSP and CCS. Assuming the existence of a link between CSP and CCS, it is possible to link mobile CSP and Pi-calculus if, following the link between mobile CSP and CSP, a formal link could be built between Pi-calculus and CCS. The definition of that link is currently undergoing and results should be published soon.

Acknowledgements

The author is grateful to Prof. Jim Woodcock for his guidance in formulating a sound theory of channel mobility. My gratitude extends to anonymous reviewers for their useful comments.

Funding Open Access funding provided by the IReL Consortium.

Appendices

A. UTP relations

Alphabet: v, v'
Signature: $\mathbf{var}\,x : T \mid \mathbf{end}\,x \mid x := e \mid SKIP \mid P \lhd b \rhd Q \mid b * P \mid P \fatsemi Q \mid P \sqcap Q \mid P_{+x} \mid \ldots$
Healthiness conditions: n/a
Semantics:

$\mathbf{var}\,x \mathrel{\widehat{=}} \exists\, x \bullet II_{\mathsf{A}}$		$\alpha(\mathbf{var}\,x) \mathrel{\widehat{=}} \mathsf{A}\setminus\{x\}$
$\mathbf{end}\,x \mathrel{\widehat{=}} \exists\, x' \bullet II_{\mathsf{A}'}$		$\alpha(\mathbf{end}\,x) \mathrel{\widehat{=}} \mathsf{A}\setminus\{x'\}$
$x :=_{\mathsf{A}} e \mathrel{\widehat{=}} (x' = e \wedge y' = y \wedge \ldots \wedge z' = z)$		$\alpha(x := e) \mathrel{\widehat{=}} \mathsf{A} \cup \mathsf{A}'$
$II_{\mathsf{A}} \mathrel{\widehat{=}} (x' = x)$	*where* $\mathsf{A} = \{x, x'\}$	$\alpha\, II_{\mathsf{A}} \mathrel{\widehat{=}} \mathsf{A}$
$P \lhd b \rhd Q \mathrel{\widehat{=}} (b \wedge P) \vee (\neg\, b \wedge Q)$	*if* $\alpha b \subseteq \alpha P = \alpha Q$	$\alpha(P \lhd b \rhd Q) \mathrel{\widehat{=}} \alpha P$
$P(v') \fatsemi Q(v) \mathrel{\widehat{=}} \exists\, v_0 \bullet P(v_0) \wedge Q(v_0)$		$in\alpha(P(v') \fatsemi Q(v)) \mathrel{\widehat{=}} in\alpha P$
		$out\alpha(P(v') \fatsemi Q(v)) \mathrel{\widehat{=}} out\alpha Q$
$P \sqcap Q \mathrel{\widehat{=}} P \vee Q$	*if* $\alpha P = \alpha Q$	$\alpha(P \sqcap Q) \mathrel{\widehat{=}} \alpha P$
$P_{+x} \mathrel{\widehat{=}} P \wedge x' = x$		$\alpha\, P_{+x} \mathrel{\widehat{=}} \alpha P \cup \{x, x'\}$
$b * P \mathrel{\widehat{=}} \mu X \bullet ((P \fatsemi X) \lhd b \rhd II)$		
$\mu\, F \mathrel{\widehat{=}} \bigsqcap\{X \mid X = F(X)\}$		
$\bot_A \mathrel{\widehat{=}} true$		$\alpha\bot_A \mathrel{\widehat{=}} A$
$\top_A \mathrel{\widehat{=}} false$		$\alpha\top_A \mathrel{\widehat{=}} A$

Laws of operators [HoaHe98, Chap. 2]:

Conditional	**Sequential composition**
$P \lhd b \rhd P = P$	$(x := e) = (x, y := e, y)$
$P \lhd b \rhd Q = Q \lhd \neg\, b \rhd P$	$(x, y, z := e, f, g) = (y, x, z := f, e, g)$
$(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$	$(x := e \fatsemi x := f(x)) = (x := f(e))$
$P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$	$x := e \fatsemi (P \lhd b(x) \rhd Q) = (x := e \fatsemi P) \lhd b(e) \rhd (x := e \fatsemi Q)$
$P \lhd true \rhd Q = P = Q \lhd false \rhd P$	$P \fatsemi II_{\alpha\, P} = P = II_{\alpha\, P} \fatsemi P$
$(P\; op\; Q) \lhd b \rhd (R\; op\; S) = (P \lhd b \rhd R)\; op\; (Q \lhd b \rhd S)$	
$(P \fatsemi Q) \fatsemi R = P \fatsemi (Q \fatsemi R)$	
$(P \lhd b \rhd Q) \fatsemi R = (P \fatsemi Q) \lhd b \rhd (Q \fatsemi R)$	
Assignment	**Skip**
$(x := e) = (x, y := e, y)$	$P \fatsemi II_{\alpha\, P} = P = II_{\alpha\, P} \fatsemi P$
$(x, y, z := e, f, g) = (y, x, z := f, e, g)$	
$(x := e \fatsemi x := f(x)) = (x := f(e))$	
$x := e \fatsemi (P \lhd b(x) \rhd Q) = (x := e \fatsemi P) \lhd b(e) \rhd (x := e \fatsemi Q)$	
Internal choice	**Variable (un)declaration**
$P \sqcap Q = P \sqcap Q$	$\mathbf{var}\,x \fatsemi \mathbf{var}\,y = \mathbf{var}\,y \fatsemi \mathbf{var}\,x \fatsemi \mathbf{var}\,x, y$
$(P \sqcap Q) \sqcap R = P \sqcap (Q \sqcap R)$	$\mathbf{end}\,x \fatsemi \mathbf{end}\,y = \mathbf{end}\,y \fatsemi \mathbf{end}\,x \fatsemi \mathbf{end}\,x, y$
$P \sqcap P = P$	$\mathbf{var}\,x \fatsemi \mathbf{end}\,y = \mathbf{end}\,y \fatsemi \mathbf{var}\,x$ provided x and y are distinct
$(P \sqcap Q) \sqcap R = (P \sqcap Q) \sqcap (Q \sqcap R)$	If T is the type of x, then $\mathbf{var}\,x = \bigsqcap\{\mathbf{var}\,x := k \mid k \in T\}$
$P \lhd b \rhd (Q \sqcap R) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$	
$P \sqcap (Q \lhd b \rhd R) = (P \sqcap Q) \lhd b \rhd (P \sqcap R)$	$\mathbf{var}\,x \fatsemi \mathbf{end}\,x = II$
$(P \sqcap Q) \fatsemi R = (P \fatsemi Q) \sqcap (Q \fatsemi R)$	$\mathbf{end}\,x \fatsemi \mathbf{var}\,x := e = (x := e)$ provided x does not occur in e
$P \fatsemi (Q \sqcap R) = (P \fatsemi Q) \sqcap (P \fatsemi R)$	$(x := e \fatsemi \mathbf{end}\,x) = \mathbf{end}\,x$
Chaos	**Miracle**
$[P \Rightarrow \bot]$ *for all P*	$[\top \Rightarrow P]$ *for all P*

B. UTP CSP

Alphabet : $v, v', \mathbf{o}, \mathbf{o'}$, where $o \in \{ok, wait, tr, ref\}$
Signature: $\mathbf{var}\, x : T \mid \mathbf{end}\, x \mid x := e \mid SKIP \mid P \triangleleft b \triangleright Q \mid b * P \mid P \mathbin{;} Q \mid P \sqcap Q \mid P_{+x} \mid P_{+B} \mid a \rightarrow P \mid f(P) \mid P \parallel Q \mid P \Box Q \mid P \setminus X \mid P \interleave Q \mid P \triangle Q \mid \ldots$
Healthiness conditions: $\boldsymbol{R1}, \boldsymbol{R2}, \boldsymbol{R3}, \boldsymbol{CSP1}, \boldsymbol{CSP2}$
Semantics:

$$(x := e) \;\widehat{=}\; \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge x' = e \wedge v' = v)$$

$$SKIP \;\widehat{=}\; \exists\, ref \bullet II_R$$

$$II_R \;\widehat{=}\; \begin{pmatrix} (ok' = ok \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge v' = v) \\ \triangleleft\, ok \triangleright tr \le tr' \end{pmatrix}$$

$$STOP \;\widehat{=}\; \mathbf{R}(wait := true)$$

$$CHAOS \;\widehat{=}\; \mathbf{R}(true)$$

$$P(\mathbf{o'}, v') \mathbin{;} Q(o, v) \;\widehat{=}\; \exists\, \mathbf{o_0}, v_0 \bullet P(\mathbf{o_0}, v_0) \mathbin{;} Q(\mathbf{o_0}, v_0)$$

$$P[e/x] \;\widehat{=}\; x := e \mathbin{;} P$$

$$c_\perp \;\widehat{=}\; II_R \triangleleft c \triangleright CHAOS$$

$$c^\top \;\widehat{=}\; II_R \triangleleft c \triangleright \top$$

$$P \parallel Q \;\widehat{=}\; P(\mathbf{o}, 1.\mathbf{o'}) \wedge Q(\mathbf{o}, 2.\mathbf{o'}) \mathbin{;} M(1.\mathbf{o}, 2.\mathbf{o}, \mathbf{o'})$$

$$M \;\widehat{=}\; \begin{pmatrix} ok' = (1.ok \wedge 2.ok) \wedge \\ wait' = (1.wait \vee 2.wait) \wedge \\ (tr' - tr) = \big((1.tr - tr) \parallel (2.tr - tr)\big) \wedge \\ ref' = (1.ref \cup 2.ref) \end{pmatrix} \mathbin{;} SKIP$$

$$P \sqcap Q \;\widehat{=}\; P \vee Q \quad , \mathcal{A}P = \mathcal{A}Q$$

$$P \Box Q \;\widehat{=}\; \boldsymbol{CSP2}\big((P \wedge Q) \triangleleft STOP \triangleright (P \vee Q)\big) \quad , \mathcal{A}P = \mathcal{A}Q$$

$$do_{\mathcal{A}}(a) \;\widehat{=}\; \Phi(a \notin ref' \triangleleft wait' \triangleright tr' = tr ^\frown \langle a \rangle)$$

$$\Phi \;\widehat{=}\; \mathbf{R} \circ and_B = and_B \circ \mathbf{R}$$

$$and_B(X) \;\widehat{=}\; B \wedge X, B \;\widehat{=}\; (tr' = tr \wedge wait') \vee tr < tr'$$

$$a \rightarrow SKIP \;\widehat{=}\; \mathbf{CSP1}(ok' \wedge do_{\mathcal{A}}(a))$$

$$a \rightarrow P \;\widehat{=}\; a \rightarrow SKIP \mathbin{;} P$$

$$ch.e \;\widehat{=}\; do_{\mathcal{A}}(ch.e)$$

$$ch?x \rightarrow P \;\widehat{=}\; \mathop{\Box}_{e \in Msg} ch.e \mathbin{;} (x := e) \mathbin{;} P$$

$$ch!y \rightarrow P \;\widehat{=}\; ch.y \mathbin{;} P$$

$$p.P \;\widehat{=}\; P[p.a \leftarrow a] \quad , \mathcal{A}(p.P) \;\widehat{=}\; \{p.a \mid a \in \mathcal{A}(P)\}$$

$$P \backslash X \;\widehat{=}\; \exists\, tra, refa \bullet \begin{pmatrix} P[tra, refa / tr', ref'] \wedge \\ tra = tr' \upharpoonright (\mathcal{A}P - X) \wedge \\ refa = ref' \cup X \end{pmatrix} \mathbin{;} SKIP$$

Laws of operators [Hoa85]:

Sequential composition	**Prefix**
$CHAOS \mathbin{;} P = CHAOS$	$a \rightarrow CHAOS \neq CHAOS$
$STOP \mathbin{;} P = STOP$	$(a \rightarrow P) \mathbin{;} Q = a \rightarrow (P \mathbin{;} Q)$
	$a \rightarrow (P \sqcap Q) = (a \rightarrow P) \sqcap (a \rightarrow Q)$
	$(a \rightarrow P \Box a \rightarrow Q) = a \rightarrow (P \Box Q)$
Internal choice	**External choice**
$P \sqcap CHAOS = CHAOS$	$P \Box CHAOS = CHAOS$
	$P \Box SKIP \sqsubseteq SKIP$
	$P \Box STOP = STOP$
	$P \Box (Q \sqcap R) = (P \Box Q) \sqcap (P \Box R)$
	$P \sqcap (Q \Box R) = (P \sqcap R) \Box (Q \sqcap R)$
	Parallel composition
	$P \parallel Q = Q \parallel P$
	$(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$
	$SKIP_A \parallel SKIP_B = SKIP_{A \cup B}$
	$STOP_A \parallel SKIP_B = SKIP_B$
	$CHAOS \parallel P = CHAOS$
	$P \parallel STOP_{\alpha P} = STOP_{\alpha P}$
	$P \parallel SKIP_{\alpha P} = SKIP_{\alpha P}$
	$(P \triangleleft b \triangleright Q) \parallel R = (P \parallel R) \triangleleft b \triangleright (Q \parallel R)$
	$(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$
	$(x := e \mathbin{;} P) \parallel Q = x := e \mathbin{;} (P \parallel Q)$
	$(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q) \quad c \in (\alpha P \cap \alpha Q)$
	$(c \rightarrow P) \parallel (d \rightarrow Q) = STOP \quad \mathit{if}\ c \neq d \quad c, d \in (\alpha P \cap \alpha Q)$
	$((x : B \rightarrow P(x)) \parallel SKIP_A) = x : (B - A) \rightarrow (P(x) \parallel SKIP_A)$
Renaming	**Hiding**
$f(STOP_A) = STOP_{f(A)}$	$P \setminus \{\} = P$
$f(a \rightarrow P) = f(a) \rightarrow f(P)$	$P \setminus A \setminus B = P \setminus A \cup B$
$f(P \parallel Q) = f(P) \parallel f(Q)$	$(P \sqcap Q) \setminus E = (P \setminus E) \sqcap (Q \setminus E)$
$f(x : B \rightarrow P(x)) = y : f(B) \rightarrow f(P(f^{-1}(y)))$	$STOP_A \setminus C = STOP_{A-C}$
$f(\mu\ X : A \bullet F(X)) = \mu\ Y : f(A) \bullet f(F(f^{-1}(Y)))$	$(P \parallel Q) \setminus E = (P \setminus E) \parallel (Q \setminus E)$
$f(g(P)) = (f \circ g)(P)$	$f(P \setminus E) = f(P) \setminus f(E)$
$f(P \sqcap Q) = f(P) \sqcap f(Q)$	$(x : B \rightarrow P(x)) \setminus C = x : B \rightarrow (P(x) \setminus C)$
	$(x : B \rightarrow P(x)) \setminus C = \sqcap_{x \in B} (P(x) \setminus C) \quad \mathit{if}\ B \subseteq C$
	$(x : B \rightarrow P(x)) \setminus C = Q \sqcap (Q \Box x : (B - C) \rightarrow P(x))$
	$(\mu\, X(a \rightarrow X)) \setminus \{a\} = CHAOS$
	$(\sqcap_n P_n) \setminus \{a\} = STOP$
	$P_{+B} \setminus B = P$

C. Healthiness conditions

In this section we present some important laws concerning healthiness conditions. Notably, for each healthiness condition we study its *idempotence*, *closure* with basic operators, and *commutativity* with other healthiness conditions.

In [CavWoo06], Cavalcanti & Woodcock carry out a systematic study of the aforementioned properties for every healthiness condition that they introduce. Adopting the methodology in [CavWoo06] would be fastidious, however. Some general results provided by Harwood and the previous two authors in [Harw08] may make such a study easier.

In [Harw08], the notion of a *conjunctive healthiness condition* allows generalising a number of important properties about healthiness conditions.

Definition C.1 (Conjunctive healthiness condition [Harw08, Def. 1]) *A healthiness condition* $\boldsymbol{CH}$ *is called* conjunctive *iff for some predicate* ψ

$$\boldsymbol{CH}(P) = P \wedge \psi \qquad \square$$

Theorem C.1 (Closure of *CH* healthy predicates)

1. *If* P *and* Q *are* $\boldsymbol{CH}$ *healthy, then* $P \wedge Q$, $P \vee Q$, *and* $P \lhd b \rhd Q$ *are* $\boldsymbol{CH}$ *healthy.*

2. *If* P *and* Q *are* $\mathbf{CH}$ *healthy, where* $\mathbf{CH}(P) = P \wedge \psi \wedge \psi'$ *(*ψ *is a condition on input variables,* ψ' *is the dashed counterpart of* ψ*), then* $P \fatsemi Q$ *is* $\boldsymbol{CH}$ *healthy.*

3. *If* F *is a monotonic function from* $\boldsymbol{CH}$ *healthy predicates to* $\boldsymbol{CH}$ *healthy predicates, then* $\mu X \bullet F(X)$ *is* $\boldsymbol{CH}$*-healthy.*

4. *If* P *and* Q *are* $\boldsymbol{CH}$ *healthy, then* $P \sqcap Q$, $P \,\Box\, Q$, $P \setminus E$, *and* $f(P)$ *are* $\boldsymbol{CH}$ *healthy.*

Proof cf. [Harw08, Thrms. 1, 2, 3], [HoaHe98, Chap. 4]. $\square$

Definition C.2 (*and*$_Q$ **[HoaHe98, Chap. 4])** *Let* $\mathbf{S}$ *denote the set of predicates of a given (UTP) theory. Let* Q *denote a predicate.*

$$and_Q \;\widehat{=}\; \lambda X : X \in \mathbf{S} \bullet Q \wedge X \qquad \square$$

Theorem C.2 *and*$_Q$ *is idempotent.*

Proof $and_Q \circ and_Q \;=\; Q \wedge (Q \wedge X) \;=\; Q \wedge X \;=\; and_Q$ $\square$

Theorem C.3 *Conjunctive healthiness conditions are instances of and*$_Q$ *viz.*

$$\boldsymbol{CH} \;=\; and_\psi$$

They are thus idempotent. Furthermore, any two conjunctive healthiness conditions are commutative since $a \wedge b = b \wedge a$. $\square$

D. Refinement in CSP

This section introduces a number of refinement concepts for CSP. Their fuller treatment can be found in [Ros10].

Let *traces*(P) denote the set of traces of a process P. Process Q *traces-refines* process P, or P is traces-refined by Q, if every trace of Q is a trace of P:

$$P \sqsubseteq_T Q \quad \textit{iff} \quad traces(P) \supseteq traces(Q)$$

Let *refusals*(P) denote the set of refusals of a process P. Let P/s denote the trace of P after a trace s. A *failure* is a pair (s, X) where $s \in traces(P)$ and $X \in refusals(P/s)$. Let *failures*(P) denote the set of all (stable) failures of P. Process Q *failures-refines* process P, or P is failures-refined by Q, if every trace and every refusal after this trace of Q are possible for P, viz., Q can neither accept nor refuse an event unless P does:

$$P \sqsubseteq_F Q \quad \textit{iff} \quad traces(P) \supseteq traces(Q) \wedge failures(P) \supseteq failures(Q)$$

Let *divergences*(P) denote the divergences of P, i.e., the set of traces after which P can diverge (and all their extensions, due to divergence strictness). Divergence strictness means that divergence is catastrophic. Process Q *failures-divergences-refines* process P, or P is failures-divergences-refined by Q, if every failure and divergence of Q are possible for P:

$$P \sqsubseteq_{FD} \quad \textit{iff} \quad failures(P) \supseteq failures(Q) \wedge divergences(P) \supseteq divergences(Q)$$

References

[Abr84] Abrial JR (1984) The mathematical construction of a program. In: Science of computer programming, vol 4. Elsevier, pp 45–86. https://doi.org/10.1016/0167-6423(84)90011-X

[BialPes09b] Bialkiewicz J-A, Peschanski F (2009) A denotational study of Mobility. In: Communicating process architectures, vol 67. IOS Press, pp 239–261. https://doi.org/10.3233/978-1-60750-065-0-239

[Borea98] Boreale M (1998) On the expressiveness of internal mobility in name-passing calculi. In: Theoretical computer science, vol 195. Elsevier, pp 205–226. https://doi.org/10.1016/S0304-3975(97)00220-X

[Broy93] Broy M, Dederichs F, Dendorfer C, Fuchs M, Gritzner TF, Weber R (1993) The design of distributed systems—an introduction to FOCUS (revised), Technical Report, University of Munich. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.30.5237&rep=rep1&type=pdf

[CavWoo06] Cavalcanti A, Woodcock J (2006) A tutorial introduction to CSP in unifying theories of programming. In: Refinement techniques in software engineering, Springer. pp 220–268. https://doi.org/10.1007/11889229_6

[Ek16a] Ekembe Ngondi G (2016) Unifying theories of mobile channels. In: Proceedings 17th international workshop on refinement, EPTCS vol 209. pp 24–39. https://doi.org/10.4204/EPTCS.209.3

[Ek16b] Ekembe Ngondi G, Woodcock J (2016) UTP semantics of reactive processes with continuations. In: Proceedings of international symposium on UTP, LNCS vol 10134. pp 114–133. https://doi.org/10.1007/978-3-319-52228-9_6

[Ek16] Ekembe Ngondi G (2016) Denotational semantics of mobility in unifying theories of programming (UTP), Ph.D. Thesis, University of York. etheses.whiterose.ac.uk/ek16.pdf

[Giu12] Giunti M, Palamidessi C, Valencia FD (2012) Hide and new in the Pi-Calculus. In: Proceedings of joint workshop on expresiveness/structure operation semantics, EPTCS vol 89. pp 65–80. https://doi.org/10.4204/EPTCS.89

[Fug98] Fuggetta A, Picco GP, Vigna G (1998) Understanding code mobility. IEEE Trans Softw Eng 24:342–361. https://doi.org/10.1109/32.685258

[GroSto99] Grosu R, Stolen K (1999) Stream based specification of mobile systems. In: Formal aspects of computing, vol 13. Springer, pp 1–31. https://doi.org/10.1007/PL00003937

[Harw08] Harwood W, Cavalcanti A, Woodcock J (2008) A theory of pointers for the UTP. In: International colloquium on theoretical aspects of computing, LNCS, vol 5160. pp 141–155. https://doi.org/10.1007/978-3-540-85762-4_10

[Hoa85] Hoare T (1985) Communicating sequential processes. Prentice-Hall

[HoaHe98] Hoare T, He Jifeng, (1998) Unifying theories of programming. Prentice-Hall

[HoaOh08] Hoare T, O'Hearn P (2008) Separation logic semantics for communicating processes. In: Proceedings of 1st international conference on foundations of information, computer and software ENTCS vol 212. pp 3–25, Elsevier. https://doi.org/10.1016/j.entcs.2008.04.050

[McEw10] McEwan A, Woodcock J (2010) Unifying theories of interrupts. In: Proceedings of international symposium on UTP, LNCS, vol 5713. Springer, pp 122–141. https://doi.org/10.1007/978-3-642-14521-6_8

[Mil99] Milner R (1999) Communicating and mobile systems: the pi-calculus. Cambridge University Press

[Ros98] Roscoe AW (1998) The theory and practice of concurrency. Prentice-Hall

[Ros10] Roscoe AW (2010) Understanding concurrent systems, Chap. 20, §20.3. pp 497–506, Prentice-Hall. https://doi.org/10.1007/978-1-84882-258-0

[Ros10a] Roscoe AW (2011) On the expressiveness of CSP, draft. http://www.cs.ox.ac.uk/files/1383/expressive.pdf

[Ros10b] Roscoe AW (2010) CSP is expressive enough for Pi. In: 'Reflections on the Work of C.A.R. Hoare', history of computing 2010, pp 371–404. https://doi.org/10.1007/978-1-84882-912-1_16

[Sang96] Sangiorgi D (1996) π-calculus, Internal mobility and agent-passing Calculi. In: Theoretical computer science, vol 167. Elsevier, pp 235–274. https://doi.org/10.1016/0304-3975(96)00075-8

[SchTre07] Schneider S, Treharne H, Vajar B (2007) Introducing mobility into CSP||B. In: Automated verification of critical systems (AVoCS). epubs.surrey.ac.uk/avocs07.pdf

[Sto99] Stølen K (1999) Specification of dynamic reconfiguration in the context of input/output relations. In: International conference on FMOODS, IFIPAICT, vol 10. Springer, pp 259–272. https://doi.org/10.1007/978-0-387-35562-7_20

[Tang04b] Tang X, Woodcock J (2004) Towards mobile processes in UTP. In: Proceedings of 2nd international conference SEFM, vol 1. IEEE, pp 44–53. https://doi.org/10.1109/SEFM.2004.10045

[Vaj09] Vajar B, Schneider S, Treharne H (2009) Mobile CSP||B. In: AVoCS, electronic communication of the EASST. https://doi.org/10.14279/tuj.eceasst.23.338

[Wel04] Welch PH, Barnes FRM (2004) Communicating mobile processes - introducing occam-pi. In: Communicating sequential processes: the first 25 years, symposium on the occasion of 25 years of CSP. pp 175–210. https://doi.org/10.1007/11423348_10

[WelBa08] Welch PH, Barnes FRM (2008) A CSP model for mobile channels. In: Communicating process architectures (CPA), vol 66. IOS Press, pp 17-33. https://doi.org/10.3233/978-1-58603-907-3-17

[Wood15] Woodcock J, Wellings AJ, Cavalcanti A (2015) Mobile CSP. In: Brazilian symposium on formal methods, LNCS, vol 9526. Springer, pp 39–55. https://doi.org/10.1007/978-3-319-29473-5_3

Received 7 February 2019
Accepted in revised form 14 March 2021 by Eerke Albert Boiten
Published online 27 May 2021